4-2020

# Toward Data-Driven Discovery of Software Vulnerabilities

Nuthan Munaiah
nm6061@rit.edu

# Toward Data-Driven Discovery of Software Vulnerabilities

by

Nuthan Munaiah

A dissertation submitted in partial fulfillment of the
requirements for the degree of
**Doctor of Philosophy**
**in Computing and Information Sciences**

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
April 2020

Toward Data-Driven Discovery of Software Vulnerabilities

by

Nuthan Munaiah

**Committee Approval:**
We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

---

Dr. Andrew Meneely — Date
Dissertation Advisor

---

Dr. Naveen Sharma — Date
Dissertation Committee Member

---

Dr. Ernest Fokoué — Date
Dissertation Committee Member

---

Dr. Pradeep K. Murukannaiah — Date
Dissertation Committee Member

---

Dr. Sharon Mason — Date
Dissertation Defense Chairperson

**Certified by:**

---

Dr. Pencheng Shi — Date
Ph.D. Program Director, Computing and Information Sciences

Toward Data-Driven Discovery of Software Vulnerabilities

by

Nuthan Munaiah

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences
Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
**Doctor of Philosophy Degree**
at the Rochester Institute of Technology

## Abstract

Over the years, Software Engineering, as a discipline, has recognized the potential for engineers to make mistakes and has incorporated processes to prevent such mistakes from becoming exploitable vulnerabilities. These processes span the spectrum from using unit/integration/fuzz testing, static/dynamic/hybrid analysis, and (automatic) patching to discover instances of vulnerabilities to leveraging data mining and machine learning to collect metrics that characterize attributes indicative of vulnerabilities. Among these processes, metrics have the potential to uncover systemic problems in the product, process, or people that could lead to vulnerabilities being introduced, rather than identifying specific instances of vulnerabilities. The insights from metrics can be used to support developers and managers in making decisions to improve the product, process, and/or people with the goal of engineering secure software.

Despite empirical evidence of metrics' association with historical software vulnerabilities, their adoption in the software development industry has been limited. The level of granularity at which the metrics are defined, the high false positive rate from models that use the metrics as explanatory variables, and, more importantly, the difficulty in deriving actionable intelligence from the metrics are often cited as factors that inhibit metrics' adoption in practice. Our research vision is *to assist software engineers in building secure software by providing a technique that generates scientific, interpretable, and actionable feedback on security as the software evolves*. In this dissertation, we present our approach toward achieving this vision through (1) systematization of vulnerability discovery metrics literature, (2) unsupervised generation of metrics-informed security feedback, and (3) continuous developer-in-the-loop improvement of the feedback.

We systematically reviewed the literature to enumerate metrics that have been proposed and/or evaluated to be indicative of vulnerabilities in software and to identify the validation criteria used to assess the decision-informing ability of these metrics. In addition to enumerating the metrics, we implemented a subset of these metrics as containerized microservices. We collected the metric values from six large open-source projects and assessed metrics' generalizability across projects, application domains, and programming languages. We then used an unsupervised approach from literature to compute threshold values for each metric and assessed the thresholds' ability to classify risk from historical vulnerabilities. We used

the metrics' values, thresholds, and interpretation to provide developers natural language feedback on security as they contributed changes and used a survey to assess their perception of the feedback. We initiated an open dialogue to gain an insight into their expectations from such feedback. In response to developer comments, we assessed the effectiveness of an existing vulnerability discovery approach—static analysis—and that of vulnerability discovery metrics in identifying risk from vulnerability contributing commits.

# Acknowledgments

My dissertation would seem incomplete without acknowledging the people who have supported me all these years.

I want to begin by thanking my advisor, Dr. Meneely, for having believed in me, and for having motivated me to pursue research that I was truly passionate about. I can say with absolute certainty that I am a better researcher and, more importantly, a better person now than when I started, thanks, in no small part, to my advisor.

I want to thank members of my committee, Dr. Sharma, Dr. Fokoué, and Dr. Murukannaiah, who have always provided the support I needed when I needed it the most. The last few years have been incredibly difficult and I could not have asked for a better committee. In addition to my committee, I want to thank Dr. Shi who has always been there if I ever needed an unvarnished assessment of a predicament. I have had many candid conversations with Dr. Shi and I truly believe he possesses an innate ability to recognize ones abilities better than they do.

I have been fortunate to have met, and collaborated with, some amazing researchers. I have learnt so much through my conversations and collaborations with these researchers that I continue to seek their counsel. I want to thank Dr. Cecilia Ovesdotter Alm, Dr. Christian Newman, Dr. Daniel Krutz, Dr. Emily Prud'hommeaux, Dr. Jay Yang, Dr. Josephine Wolff, Dr. Justin Pelletier, Dr. Laurie Williams, Dr. Linwei Wang, Dr. Mehdi Mirakhorli, Dr. Mei Nagappan, Dr. Mohamed Wiem Mkaouer, Dr. Yang Yu, Akond, Chris Theisen, Chris Horn, Craig, Felivel, Iván, Kevin, and Steven.

In the last six years, I have had the pleasure of meeting the incredible people who tirelessly work in the background to support our academic endeavors. I want to thank Tracy and Amanda for making sure I always got paid; Lorrie Jo for making sure I got into the classes I wanted, I had a conference room when I needed one, and for always being there if I ever wanted to simply have a conversation; Min-Hong for helping me navigate the graduation process throughout the last year; Bridgette, Carrie, Chelsea, and Dawn for helping me with anything I needed during my time in the Department of Software Engineering; and Charles, Kurt, Arnela, and all the System Administrators at Research Computing for providing me computing resources needed to conduct my research.

I want to thank all my friends for providing a sense of normalcy to my life all these years: Aditya, Adriana, Akshay, Anudeep, Anup, Ben, Bruce, Danielle, Erika, Erin, Farhad, Gordon, Harold, Joanna, Joe, John, Jwala, Khaled, Liz, Madhuresh, Muhammad, Nataraj, Niranjana, Nirmal, Niren, Nishi, Palak, Radhika, Renee, Sarah, Shabana, Selma, Smita, Srinivasa, Teja, Vasudha, and Waleed. I want to apologize to anyone I may have forgotten to explicitly acknowledge here but know that if we have ever talked about research, you have helped me by listening, and for that, I thank you.

In closing, I want acknowledge the funding agencies who have supported our work throughout the years. Our research has been supported by the U.S. National Security Agency (NSA) Science of Security Lablet (SoSL) at North Carolina State University (H98230-17-D-0080/2018-0438-02), the U.S. National Science Foundation CyberCorps® Scholarship for Service Program (1433736), and the Small Business Innovation Research (SBIR) Program of the U.S. Defense Advanced Research Projects Agency (DARPA).

*To my mother, Indira, and my father, Munaiah, who have always encouraged and supported my aspirations.*

*To my brother and sister-in-law, Naveen and Ullasini, and my wonderful niece, Charvi, who have provided a sense of normalcy to my life these last few years.*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

As more aspects of our daily lives depend on technology, the software that supports this technology must be secure. We, as users, automatically assume the software we use to always be available to serve our requests while preserving the confidentiality and integrity of our information. Unfortunately, incidents involving catastrophic software vulnerabilities such as Heartbleed (in OpenSSL), Stagefright (in Android), and EternalBlue (in Windows) have made abundantly clear that software systems, like all other engineered creations, are prone to mistakes.

Over the years, Software Engineering, as a discipline, has recognized the potential for engineers to make mistakes and has incorporated processes to prevent such mistakes from becoming exploitable vulnerabilities. The Trustworthy Computing Memo by Bill Gates [35] is an instance of a software development organization highlighting the importance of security in the products they engineer. The memo led to the creation of Security Development Lifecycle (SDL) [46] which emphasized the need to prioritize security during the entire development lifecycle of software. More importantly, the SDL encouraged software engineers to have a *conversation* about software security. For instance, threat modeling—a design activity prescribed in the SDL—prompts software engineers to have a conversation about securing assets (such as processes, and data stores), that the software being designed will likely have access to, against potential threats from attackers. In addition to improving the overall quality and reliability of software, conversation about security helps software engineers to develop an *attacker mindset*, which, as researchers [117, 44, 75] have argued, is an essential skill.

The demand for vulnerability exploits, and the existence of a business model to support their trade [11, 78], provides monetary benefits for the attackers to invest considerable effort to uncover exploitable vulnerabilities in software. In the current situation where developers and attackers are engaged in a seemingly never-ending battle, the attackers seem to be the innovators. We, as software engineering researchers, must arm developers with the necessary arsenal to engineer software that is potentially impregnable by such malicious attackers. In an effort to provide software engineers with the means of discovering potential security vulnerabilities, researchers have proposed a plethora of metrics defined on product, process, and people. Despite the metrics being shown to be effective at assisting in the discovery of historical vulnerabilities, their adoption in the software development industry has been

limited. The high false positive rate from models that use the metrics as explanatory variables [91] and difficulty in deriving actionable intelligence from the metrics [32] are often cited as the factors inhibiting metrics' adoption. While the apprehension on part of the software development industry is justified, the utility of the metrics as a promoter of conversation about security is worth exploring.

The art of engineering software is inherently collaborative [152]; as stakeholders with diverse skill sets come together to specify, manage, develop, test, and release software, conversations within a team of software engineers play a crucial role in facilitating the progress of a software project. In the development phase of a project, however, software engineers rely on feedback from tools such as integrated development environments that aggregate errors and warnings from compiling, building, testing, and/or statically analyzing the source code. While feedback conveyed by these tools is useful, developers have difficulty interpreting the feedback as Johnson *et al.* found in their study of static analysis tools [52]. We argue that feedback about security requires an attacker mindset to exhaustively evaluate and that such feedback is most effective if it prompts a conversation about security. In effect, we are proposing the extension of the popular anecdote dubbed Linus' Law stated by Eric Raymond as "Given enough eyeballs, all bugs are shallow" [121] to feedback on security.

In a tool-assisted approach to software development today, there are several opportunities for providing feedback in a way that it prompts a conversation. Such opportunities are appropriate for providing feedback on security as a conversation about security is essential to inculcate an attacker-defender culture among software engineers. The process of code review is a prime example of an opportunity for providing feedback on security as code reviews are intended to uncover quality concerns through collaboration. Furthermore, many developers perceive code reviews to aid in knowledge transfer and increasing team awareness [70], which is an added benefit. We can leverage security vulnerability discovery metrics proposed in research literature to provide automatically generated feedback on security during processes such as reviewing code or triaging issues to highlight the security vulnerability potential.

Our research vision is *to assist software engineers in building secure software by providing a technique that generates scientific, interpretable, and actionable feedback on security as the software evolves.* In accomplishing the research vision, we hope to influence the discipline of software engineering through:

- the systematization of metrics that are known to be effective in assisting software engineers to discover vulnerabilities,
- the development of a usable approach to provide security feedback to software engineers, and
- the reference implementation of a platform that leverages the vulnerability discovery metrics to provide software engineers with automatically generated feedback on security.

In demonstrating the utility of the metrics as agents of feedback on security, we will be one step closer to bringing the vulnerability discovery knowledge from research literature into mainstream software engineering.

## 1.1 Overview

In this section, we provide a narrative overview of chapters that compose the dissertation. We, however, begin the narration by discussing Chapter 4 which was the catalyst that

prompted us to define our research vision. In Chapter 4, we propose fine-grained vulnerability discovery metrics and describe the (conventional) approach to reasoning about the utility of these metrics in discovering vulnerabilities. We show that the metrics are statistically significantly associated with historical vulnerabilities in two large open-source projects and that the prediction model built using the metrics as features considerably outperforms existing vulnerability prediction models. While the prediction model evaluated seems utilitarian relative to existing models, the absolute performance of the model leaves a lot to be desired. Once the study described in Chapter 4 was published [100], we asked of the metrics—Do the metrics, that we argue as being predictive of vulnerabilities, actually help developers discover vulnerabilities?—and this question became the catalyst that prompted us to define our research vision.

The central position of this dissertation is that vulnerability discovery metrics have the potential to assist developers in engineering securing software and that this potential goes beyond the mere performance of a prediction model that uses the metrics as features. The overall approach (depicted in Figure 1.1) to systematically achieve the research vision involves three steps: (1) systematization of vulnerability discovery knowledge, (2) unsupervised generation of metrics-informed security feedback, and (3) continuous developer-in-the-loop improvement of approach.

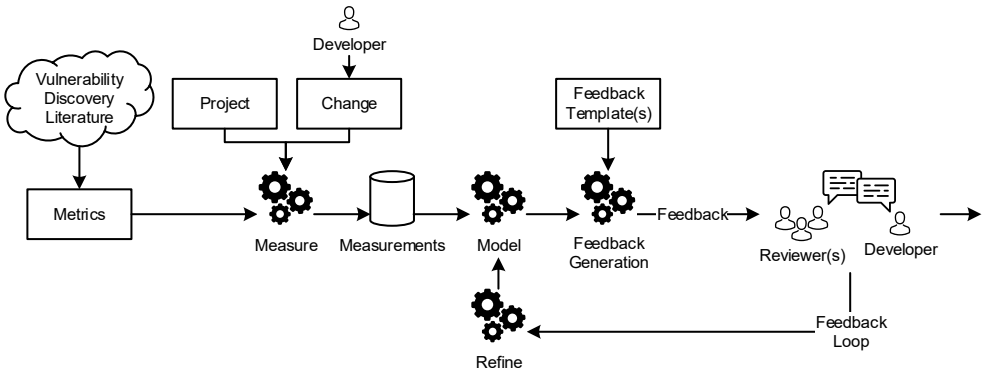



Figure 1.1: Pictorial representation of the approach to systematically achieve the research vision

The first step, described in Chapter 5, is to enumerate the vulnerability discovery metrics in the literature through a systematic literature review. We addressed the research questions that follow in the systematic literature review.

**RQ 1 - Enumeration**

What metrics have been proposed to discover security vulnerabilities in software?

**RQ 2 - Validation**

How have researchers evaluated the decision-informing ability of the metrics to discover security vulnerabilities in software?

While systematization of the vulnerability discovery metrics from the literature is necessary, implementing the metrics is an essential prerequisite to the subsequent steps of our approach. We decided to use an unconventional approach to implementing the metrics and this decision was guided by one, and only one, goal: convenience. We chose to implement the metrics as containerized microservices allowing researchers and practitioners to collect metrics from any project with minimal effort. We hope the convenience lowers the barrier to entry to the use of metrics so much so that researchers are encouraged to consider replicating the metrics and practitioners are encouraged to use the metrics in practice. In addition to the aforementioned research questions, we also addressed secondary research questions that follow as part of the literature review.

**RQ 3 - Generalizability**

Are vulnerability discovery metrics similarly distributed across projects?

**RQ 4 - Thresholds**

Are thresholds of vulnerability discovery metrics effective at classifying risk from vulnerabilities?

The second step, described in Chapter 6, is to use the metrics in the context of a model to derive vulnerability discovery insights. The insights gained from the model are used to generate feedback using natural language generation techniques. The automatically generated feedback is communicated to the developer in a context (e.g. code review) where it is likely to prompt a conversation about security. We address the research questions that follow to evaluate the feedback provided to developers.

**RQ 5 - Feedback**

How is feedback informed by insights from vulnerability discovery metrics perceived by developers?

**RQ 6 - Expectations**

What are developers' expectations from vulnerability discovery metrics?

**RQ 7 - Effectiveness**

How effective are existing vulnerability discovery approaches?

**RQ 8 - Utility**

Is there a utility for vulnerability discovery metrics?

The third step, involving the continuous developer-in-the-loop feedback on our approach, is used to inform the first and second steps as necessary. As a result, there is no separate chapter dedicated to the third step in our approach. In fact, research questions RQ 6, RQ 7, and RQ 8 were formulated as a result of the developer-in-the-loop feedback that prompted us to adapt our approach while also addressing certain fundamental research questions.

## 1.2 Contributions

In this section, we enumerate our tangible contributions to the community (via data sets, tools, and services) and knowledge (via publications). While some of the contributions are directly related our research vision, others are from complementary research projects conducted during the program. The contributions enumerated here is a subset so chosen to highlight those that have had notable impact in community and/or knowledge.

### 1.2.1 Data Sets, Tools, and Services

We published several data sets, tools, and services as part of multiple research projects. The enumeration of contributions follows with those not related to the domain of security highlighted with a super-scripted asterisk (*Non-security contribution*$^*$). All projects contribute to the research in and/or the practice of Software Engineering.

1. SAMARITAN Metric Services - The SAMARITAN metric services, designed and developed as part of the work described in this dissertation, are a key contribution to the community. The services are implementations of the vulnerability discovery metrics being released to encourage researchers to replicate and practitioners to use. The specifics of the services are described in Appendix A.
2. SAMARITAN Website - The SAMARITAN website, available at `https://samaritan.github.io/`, complements the SAMARITAN metric services by being a source of information about the metrics. Each metric has its own page on the website providing information such as the definition, the academic publications that have validated the metric, the consensus within the community about the metrics' relationship with vulnerabilities, the distribution of the metric in example open-source projects, and the thresholds values that delineates the risk levels computed in an unsupervised way. We hope the website makes vulnerability discovery metric knowledge accessible thus bridging the gap between research and practice.
3. SAMARITAN `archeogit` - `archeogit` in an open-source utility, available on GitHub at `https://github.com/samaritan/archeogit/`, developed to excavate information from `git` repositories. The `archeogit blame` command implements a heuristic-based algorithm (similar to the one used by Meneely *et al.* [85] and Perl *et al.* [115]) to identify commit(s) that likely contributed the bug which was later fixed. As we describe in Chapter 6, `archeogit blame` was essential to identifying vulnerability contributing commits. The utility is also being used by students in the Engineering Secure Software (SWEN-331) course to support the data curation for the Vulnerability History Project (`https://vulnerabilityhistory.org/`).
4. Attack Surface Meter - The Attack Surface Meter, available at `https://pypi.org/project/attacksurfacemeter/`, is a tool developed to collect fine-grained attack surface metrics from a C/C++ project. The tool was developed to support the research project described in Chapter 4 and has been cited to have been used in replication studies (see study by Theisen [146]).
5. GHTorrent `reaper` - The GHTorrent `reaper` is an open-source utility, available at `https://github.com/reporeapers/reaper/`, designed and developed to collect metrics from repositories hosted by GitHub. The utility, designed to be scalable, was used to analyze 1.8 million repositories from GitHub in less than three calendar months'

time. The metrics collected from the analyzed repositories was used to identify those repositories that likely contained engineered software projects. The supporting website `https://reporeapers.github.io/` disseminates the data set for researchers to use in their search for GitHub repositories to study. As on April 2020, `reaper` has 66 stargazers on GitHub. Furthermore, `reaper` and the data set curated using `reaper` are featured on *awesome-msr* (`https://github.com/dspinellis/awesome-msr`), a curated list of data sets and tools in Empirical Software Engineering.

### 1.2.2 Publications

We have published eight notable papers as part of multiple research projects conducted during the program. The enumeration of publications follows with all but one, highlighted with super-scripted asterisk (*Non-security contribution*$^*$), contributing to the domain of software security.

1. Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project in the Empirical Software Engineering journal [104]

   In this study, we investigated the co-location of bugs and vulnerabilities in two large open-source projects: Chromium and Apache httpd. We evaluated the hypothesis that files with lots of bugs must be vulnerable. However, we found evidence, consistent across both projects, that bugs and vulnerabilities are likely to be empirically dissimilar groups. The implication of this study is that vulnerabilities are unlikely to be discovered using approaches that have been effective in discovering bugs and that approaches targeted toward vulnerability discovery are necessary. The study lends credence to the work presented in this dissertation.

2. Beyond the Attack Surface: Assessing Security Risk with Random Walks on Call Graphs in the Proceedings of the 2016 ACM Workshop on Software PROtection (SPRO) [100]

   In this study, we proposed and validated two fine-grained metrics to discover vulnerabilities in software. The study is included as Chapter 4 in this dissertation.

3. Natural Language Insights from Code Reviews that Missed a Vulnerability in the Proceedings of the 2017 International Symposium on Engineering Secure Software and Systems (ESSoS) [106]

   In this study, we investigated the linguistic characteristics of developer conversations in code reviews that likely missed a vulnerability. In characterizing messages in a code review that likely missed a vulnerability, we hoped to provide interventional feedback to developers if a future code review message exhibited similar characteristics. We proposed and validated a set of metrics using over 3.9 million messages posted by Chromium developers and found statistically significant associations between the metrics and code reviews that likely missed a vulnerability.

4. Vulnerability Severity Scoring and Bounties: Why the Disconnect? in the Proceedings of the 2016 ACM International Workshop on Software Analytics (SWAN) [101]

   In this study, we investigated the economic validity of the standard metric to quantify vulnerability severity—*base score* as defined by the Common Vulnerability Scoring System [80]. We hypothesized that base score, being a measure of vulnerability severity, must be correlated with the monetary value of the vulnerability. The intuition

behind the hypothesis is that a severe vulnerability, one that an attacker could exploit to inflict considerable damage, must have a proportionately higher monetary value. However, we found that the correlation was negligible-to-weak. We qualitatively analyzed the criteria used to assign vulnerabilities the base score and that used to assign the monetary values. We made concrete recommendations based on the limitations we uncovered in the qualitative analysis.

5. Data-driven Insights from Vulnerability Discovery Metrics in the Proceedings of the 2019 International Workshop on Data-Driven Decisions, Experimentation and Evolution (DDrEE) [102]

   In this study, we implemented and collected ten vulnerability discovery metrics and assessed their generalizability across projects. We also assessed the ability of metric thresholds, established using an unsupervised approach, to classify risk from historical vulnerabilities in Chromium. The study is included as a part of Chapter 5 in this dissertation.

6. A Domain-Independent Model for Identifying Security Requirements in Proceedings of the 2017 IEEE International Requirements Engineering Conference (RE) [95]

   In this study, we proposed and validated a One-Class Support Vector Machine to sift out security requirements in a corpus of software requirements. The novelty of the model was in its ability to identify security requirements for projects in a domain-agnostic way. Our model outperformed an existing model from prior literature which required domain-specific datasets for training.

7. Characterizing Attacker Behavior in a Cybersecurity Penetration Testing Competition in Proceedings of the 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) [96]

   In this study, we proposed using attacker behavior, characterized from data collected during a cybersecurity penetration testing competition, to aid the development of attacker mindset. We systematically identified 44 events that described the attack campaign of one of the teams participating in the competition. We used the MITRE ATT&CK™ framework to characterize and disseminate the timeline in a standardized way.

8. Curating GitHub for Engineered Software Projects in the Empirical Software Engineering journal* [105]

   In this study, we proposed an approach to filter repositories from GitHub based on nine metrics argued to measure essential attributes of an engineered software project: architecture, community, continuous integration, documentation, history, issues, license, size, and unit test. The proposed approach provided researchers in the Mining Software Repositories community an alternative to relying on the popularity of repositories on GitHub as a filtering criteria which implied that popularity correlates with quality. We analyzed over 1.8 million repositories from GitHub and showed that the proposed approach considerably outperformed the de facto approach of filtering repositories by popularity. As on April 2020, the study has 62 citations as reported by Scopus®.

Even though some of the publications are not directly included in the dissertation, they have indirectly influenced the approach used toward achieving our research vision. For

instance, the decision to use containerized microservices to implement the metrics was influenced by the inconvenience experienced by a researcher in using `reaper` to collect metrics from a few thousand GitHub repositories.

# Chapter 2

# Background

In this chapter, we provide a brief overview of the key terms related to software anomalies, software metrics, and software engineering research.

## 2.1 Software Anomalies

Software anomaly is used to refer to a wide variety of software inconsistencies such as errors, faults, failures, defects, and bugs [48]. A component is said to have a *defect* if it does not meet its requirements or specifications [49, 3.764]. For example, a component responsible for authentication and authorization in a web application is said to be defective if it does not invalidate the authentication cookie after users successfully sign out of the application. An *error* is the difference between expected and actual outcomes observed during a typical interaction with software [49, 3.1027]. For example, software is said to have an error when users see the text "Hello ," instead of "Hello {last name}, {first name}" where {last name} and {first name} are the last and first names of the user who successfully signed in to the application. A software *fault* is the manifestation of an *error* in software [49, 3.1122] which, when encountered, may cause a failure [49, 3.1122]. The term *bug* is used as a synonym for fault. For example, a developer missing input validation is an instance of an error. The missing validation error could lead to multiple faults such as division by zero if unvalidated input is used as a number in the denominator or type mismatch if unvalidated input is assigned to a variable of an incompatible type. When one of these faults is executed, users experience failures. The term *defect* is also used as a generic term to refer to fault (cause) or failure (effect) [49, 3.764].

In this research, we will restrict ourselves to a particular kind of software defect, one that violates an implicit or explicit security policy, called *vulnerability*. A key difference between vulnerabilities and other types of software anomalies is the presence of an (active) adversarial agent. In 1998, Krsul defined *software vulnerability* as "an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy." [57]. Ozment modified Krsul's definition by replacing "error" with "mistake" to capture the human element and using "implicit or explicit" to qualify "security policy" [112]. The phrases *software vulnerability* or *security vulnerability* are synonymous

with the term *vulnerability* and we use these three interchangeably. A classic example of a security vulnerability is the buffer overflow vulnerability [26] which, when exploited by an adversary, can be used to cause the software to execute malicious code.

## 2.2 Software Metrics

The ISO/IEC define a *(software) metric* as "a quantitative measure of the degree to which a system, component, or process possesses a given attribute" [49, 3.1767]. At a high-level, software metrics may be categorized into two groups: (1) *product metrics*, which measure attributes of the software and (2) *process metrics*, which measure attributes of the software development process. The attribute that a metric purports to quantify can either be internal (e.g. size) or external (e.g. security or diversity) to the product or process.

## 2.3 Software Engineering Research

In the subsections that follow, we provide a brief overview of key terms used in (empirical) software engineering research, specifically, on vulnerability discovery.

### 2.3.1 Software Repositories

Large software projects are typically supported by a plethora of tools that facilitate collaboration among software engineers. These tools use databases, commonly referred to as *repositories*, to store data. The data used in the empirical study of vulnerabilities are typically aggregated from one or more such repositories. In a typical software engineering project, one or more of the following software repositories may be in use:

- *Source Code Repository*: A source code repository has become a commonplace in most software engineering projects. Source code repositories enable developers to simultaneously contribute changes to a project by tracking the history of all files. The smallest unit of change tracked by a source code repository is called a *commit*. In addition to maintaining the contents of the commit, the source code repository also records metadata about the commit such as an unique identifier, summary of the change, date and time of the commit, email address of the developer who authored the change, email address of the developer who committed the change, files modified by the change, and number of lines added and/or removed by the change. The tool that facilitates the maintenance of a source code repository is called a version control system. `git`[1] is a popular version control system that is widely used in the open source community. GitHub[2] is an online hosting service that enables sharing of `git` repositories with the wider open source community.

- *Issue Tracking Repository*: An issue tracking repository is similar to the source code repository but it facilitates the lifecycle management of issues (bugs). An issue evolves as developers triaging the issue contribute *comments* to the conversation. In addition to the comments, the bug itself may have a variety of metadata associated with it such as the date and time the issue was opened, a summary of the issue provided by the user who opened the issue, the type of the issue, the status of the issue and

[1] `https://git-scm.com/` [2] `https://github.com/`

the developer assigned to the issue. Bugzilla[3] is a popular open source issue tracking system that is widely used in the open source community.

- *Code Review Repository*: A code review is a lightweight, tool-assisted, process used to improve the overall quality of the software as it evolves. A typical code review involves a developer (*author*) submitting changes to a set of files, called a *patchset*, for review by other developers (*reviewers*). The reviewers provide feedback using comments at the line level, file level, or review level. The author is responsible for addressing the comments and submitting a revised patchset for review. The reviewers approve the patchset if it meets their requirements. After a patchset is approved, it is automatically committed to the source code repository. Gerrit[4] is a popular open source code review system that is widely used in the open source community.

- *Vulnerability Repository*: A vulnerability repository is an authoritative source for information on historical vulnerabilities. As of this writing, the National Vulnerability Database (NVD) [1] is an example of an exemplary vulnerability repository that contains details of all vulnerabilities that have been disclosed by software vendors.

A software project that uses one or more of these software repositories tend to have traceability links between entities in different repositories to simplify historical analysis. For example, a vulnerability in the NVD is likely to have links to a bug in the issue tracking that may have been internally opened to triage the fix for the vulnerability. The bug in turn may be associated with one or more code reviews and the code reviews may be linked to commits in the source code repository.

### 2.3.2 Vulnerability Discovery

In the empirical study of historical vulnerabilities, an essential step is to identify source code entities (lines, files, functions, modules, or binaries) that have been fixed for a vulnerability. Source code entities so identified are labeled as *vulnerable* and all other entities are labeled as *neutral*. In validating the metrics being proposed to discover vulnerabilities, researchers typically separate source code entities using the label (vulnerable or neutral) and perform analysis to test if a metric is associated with historically vulnerable source code entities or if the metric can be used in a machine learning model to identify historically vulnerable source code entities. Traditionally, machine learning models used in identifying historically vulnerable source code entities take the form of a binary classifier with the metric being the *independent variable* (also known as *explanatory variable*, *feature*, or *predictor*) and label (vulnerable or neutral) being the *dependent variable* (also known as the *response variable*).

[3] `https://www.bugzilla.org/` [4] `https://www.gerritcodereview.com/`

# Chapter 3

# Related Work

The accomplishment of the research vision set forth in this dissertation borrows knowledge from prior literature in the areas of vulnerability discovery and feedback in software engineering. The two sections that follow include a brief overview of some of the relevant literature in each of these two areas. The third section provides a summary highlighting the opportunities to leverage the best of both areas to assist developers in engineering secure software.

## 3.1 Vulnerability Discovery

Since the year 2000, researchers have made considerable progress in characterizing vulnerabilities. The proactive discovery of vulnerability has been one of the primary goals in multiple studies from prior literature. The vulnerability discovery approaches proposed in prior literature span the spectrum of proactive assessment of software security from using testing, static, dynamic or hybrid analysis, (automatic) patching [130] to leveraging data mining and machine learning to collect metrics that may be used to characterize and predict vulnerabilities [39].

We restrict ourselves to vulnerability discovery approaches that leverage metrics to characterize and discover vulnerabilities because:

1. The metrics can be used to infer actionable intelligence about impending security risk rather than identify instances of vulnerabilities and
2. Some of the metrics are relatively easy to collect and may already be continually collected as part of the software development process (e.g. the CODEMINE platform at Microsoft [28])

The survey by Ghaffarian and Shahriari [39] provides a summary of existing literature on vulnerability discovery approaches that leverage metrics published after the year 2010. We summarize the prior literature here with attention devoted to the metrics used in characterizing the vulnerabilities.

The first empirical study on vulnerabilities is by Neuhaus *et al.* [110] from 2007. In this study, dependency relationships, expressed via the `#include` preprocessor directive, between

Table 3.1: Summary of relevant prior vulnerability discovery metrics literature

| Author | Metrics |
|---|---|
| Neuhaus, Stephan [110, 109] | Dependency between components and packages |
| Shin, Yonghee [135, 136, 133, 137, 134, 138] | Complexity |
| Chowdhury, Istehad [22] | Complexity, coupling, and cohesion |
| Meneely, Andrew [81, 82, 85] | Developer activity and collaboration and interactive churn |
| Zimmermann, Thomas [160] | Churn, complexity, coverage, dependency, organizational, and dependency between binaries |
| Scandariato, Riccardo [127] | Features derived using text mining |
| Walden, James [149] | Complexity, coupling, dependency, and features derived using text mining |
| Perl, Henning [115] | Churn, experience, GitHub metadata, and features derived using text mining |
| Younis, Awad [156] | Size, structure, ease of access, and communication |

components in Mozilla were purported to indicate components prone to vulnerabilities. The researchers collected historical vulnerability data and identified that 424 (4.05% of 10,452) components were modified during a fix for a vulnerability. A Support Vector Machine classifier trained and tested with the historical data was shown to predict vulnerable components in Mozilla with an average precision and recall of 0.65 and 0.45, respectively. The dependency relationship, though effective at predicting vulnerable components in Mozilla, is unlikely to yield generalizable knowledge that applies to any software project. However, being the first empirical study on predicting vulnerabilities, the study does demonstrate that vulnerabilities can indeed be predicted using metrics.

The earliest instance of using metrics collected from source code used to characterize vulnerabilities is an empirical study by Shin and Williams [135]. In this study, nine complexity metrics were assessed to be indicators of vulnerability-prone functions. The researchers collected historical fault and vulnerability data and identified that 0.5% to 16.6% functions were modified during a fix for a fault and 0.4% to 9.4% functions were modified during a fix for a vulnerability across four releases of the Mozilla JavaScript Engine. In the empirical analysis, the metrics were shown to be weakly correlated with number of vulnerabilities. However, the association analysis revealed that fault-prone functions were less complex than vulnerability-prone function, strengthening the notion that faults may be distinct from vulnerabilities. In a similar study [136], Shin and Williams assessed the capabilities of the nine complexity met-

rics as indicators of vulnerability-prone functions in the context of a machine learning model. A Logistic Regression classifier was trained and tested on historical data obtained from six releases of the Mozilla JavaScript Engine. The classifier exhibited an average accuracy, false positive rate, and false negative rate of 94.35%, 0.74%, and 86.78%, respectively. While the accuracy of the classifier is high the correspondingly high false negative rate is concerning as the classifier may be misclassifying a large portion of the vulnerable functions as neutral.

Neuhaus and Zimmermann [109] extended their earlier study [110] assessing the relationship between dependencies of a component and its vulnerability-proneness to dependencies between packages in the Red Hat distribution of the Linux Operating System. The researchers collected historical vulnerability data and identified that 1,133 packages were updated to resolve a vulnerability. A Support Vector Machine classifier trained and tested with the historical data was shown to predict vulnerable packages in Red Hat with a median precision and recall of 0.83 and 0.65, respectively, outperforming a Decision Tree classifier trained and tested on the same data. In aggregate, both studies by Neuhaus [110, 109] allude to the notion that dependency between source code entities (components or packages) is an indicator of the entity being vulnerability prone. While the classifier implemented using the dependency metric achieved reasonable precision and recall, the insight gained from the classifier is not actionable. Classifying a file as vulnerable simply because it is dependent on another is not enough information for a developer to understand the rationale behind the classification.

In more recent times, the study by Zimmermann *et al.* [160] was key in vulnerability discovery for two reasons: (1) likened the process of discovering vulnerabilities to the task of searching for a needle in a haystack and (2) was the first large-scale empirical study assessing the efficacy of five groups of classical metrics describing characteristics of source code and team organization as indicators of a vulnerability-prone binary. The metrics were empirically validated using 66 historical vulnerabilities in the Windows Vista Operating System. The empirical analysis revealed that the metrics were weakly correlated with number of vulnerabilities in the binaries. Furthermore, in assessing the effectiveness of a series of Logistic Regression classifiers using different groups of metrics, trained and tested with the historical data, the classifier that used metrics related to churn, dependency and organization exhibited the highest median precision of 0.67 and the classifier that used all five groups of metrics exhibited the highest median recall of 0.20. The study also replicated the notion—from prior studies by Neuhaus [110, 109]—that dependencies (between binaries) may be an indicator of a binary being vulnerability prone. A Support Vector Machine classifier used actual dependencies between binaries to predict a binary as vulnerable with a median precision and recall of 0.60 and 0.40, respectively. A key limitation in this study was that the empirical evaluation was at the granularity of a binary. The insight that a binary may be vulnerable is not actionable because a developer may have to review hundreds of files that are compiled into the binary to uncover potential vulnerabilities.

The metrics used to characterize vulnerabilities so far were derived from the product alone, the people (i.e. software developers) were not considered as a factor contributing to software vulnerability. Meneely and Williams [81] were the first to propose a set of metrics, called developer activity metrics, to quantify the contribution of and collaboration among software developers. The metrics were derived from developer networks constructed using information derived from version control logs. The researchers collected historical vulnerability data from Red Hat Enterprise Linux Kernel (RHEL4) and identified 205 files that were modified during a fix for a vulnerability. In the empirical analysis, the researchers found the developer activity metrics to be statistically significantly different between vulnerable and neutral files. Furthermore, two classifiers implemented using Multivariate Discriminant

Analysis and Bayesian Networks, trained and tested with the historical data, was capable of predicting vulnerable files, however, the Bayesian Networks classifier exhibited the highest precision and recall of 13.3% and 33.2%, respectively. While the precision and recall measures are not impressive, the statistically significant difference in the metric between vulnerable and neutral files alludes to the notion that the way in which developers contribute to and collaborate in a project does indeed have an impact on the security of the software. In a follow-up study, Meneely and Williams [82] replicated the study using historical vulnerability data from the PHP Programming Language, Wireshark Network Protocol Analyzer and updated Red Hat Enterprise Linux Kernel (RHEL4) and found similar results as before. The classifiers implemented using Bayesian Networks performed better with an average precision and recall of 20.73% and 72.27%, respectively. The key benefit of using developer activity metrics is that the they afford a deeper insight into the software development process. The insights can be used to inform organizational and managerial changes which are relatively easier to implement than behavioral changes to developers. Furthermore, developer activity metrics, being process metrics, have certain characteristics that make them better than product metrics at certain tasks [120].

The code complexity metrics used to characterize vulnerabilities so far were statically collected i.e. the software was not required to be executed to collect the metrics. In arguing that metrics collected during the execution of software are likely to indicate location of vulnerable code, Shin and Williams [137] proposed a set of three metrics categorized as execution complexity metrics. The researchers collected historical vulnerability data from Mozilla Firefox web browser and Wireshark Network Protocol Analyzer and identified 301 and 181 files that were modified during a fix for a vulnerability, respectively. In the empirical analysis of the execution complexity metrics and other statically collected metrics, the researchers found statistically significant difference in most of the metrics between vulnerable and neutral files. The execution complexity metrics, however, were not consistently statistically significant between Mozilla Firefox and Wireshark. The Logistic Regression classifier trained and tested with the historical data performed the best, in terms of reduction in cost of inspection, when using only the execution complexity metrics.

While the developer activity metrics proposed by Meneely and Williams [81, 82] were empirically shown to be indicators of vulnerability-prone files, the analysis was conducted in isolation from other vulnerability discovery metrics. Shin *et al.* examined the utility of the developer activity metrics proposed by Meneely and Williams [81, 82], analyzing the metrics in concert with other classical metrics such as code complexity and code churn [133]. The researchers collected historical vulnerability data for Mozilla Firefox web browser and Red Hat Enterprise Linux Kernel (RHEL4) and identified files that were modified during a fix for a vulnerability. In the empirical analysis of developer activity metrics and other conventional metrics of code complexity and code churn, the researchers found that 27 of the 28 metrics were statistically significantly different between vulnerable and neutral files. Furthermore, a Logistic Regression classifier using all three groups of metrics performed the best with a mean probability of detection of 0.84 and mean probability of false positive of 0.24. While the mean precision of the model was low at a mere 0.05, the classifier outperformed random selection by lowering the cost of inspection by 71% and 28% at the file and line levels, respectively.

In an attempt to evaluate the vulnerability discovery capabilities of complexity, coupling, and cohesion metrics that were shown to have utility in defect discovery, Chowdhury and Zulkernine [22] conducted an empirical study using historical vulnerability data collected from five releases of the Mozilla Firefox web browser. In the study, researchers found statistically significant moderate-to-strong positive correlation between 17 metrics and the

number of vulnerabilities in a file. In assessing the capability of the metrics as indicators of vulnerability-prone files in a machine learning model, a Decision Tree classifier trained and tested on historical vulnerability data exhibited an average precision and recall of 71.82% and 74.22%, respectively. The study was instrumental in highlighting the possibility of using coupling and cohesion metrics, in addition to traditional complexity metrics, as indicators of vulnerability-prone files.

Vulnerabilities and faults are similar in that both can manifest as a result of a mistake that the software developer made during development of software. Shin and Williams [138] attempted to reap the benefits of this similarity by attempting to use models traditionally used to discover faults in software to discover vulnerabilities. A total of 25 metrics, spanning three types (code complexity, code churn, and fault history), traditionally known to be indicators of faults were used in the implementation of a series of Logistic Regression classifiers. The classifiers were trained and tested using historical fault and vulnerability data obtained from Mozilla Firefox web browser. In the empirical analysis, researchers found that the classifier trained with historical fault data exhibited comparable performance to that of a classifier trained with historical vulnerability data in identifying vulnerability-prone files. The similarity in classifier performance implies that, in the absence of historical vulnerability data, historical fault data may be used to identify vulnerability-prone files.

In almost all of the studies summarized so far, code complexity metrics seem to be widely studied as an indicator of vulnerability. The empirical analysis conducted by Moshtari *et al.* [94] further strengthens the support for code complexity metrics being effective indicators of vulnerabilities by (1) replicating the empirical study by Shin *et al.* [133] using seven classification techniques and (2) performing a cross-project analysis using five open source projects. In the replication study, the researchers found that a classifier implemented using the IBK technique outperformed the Logistic Regression classifier proposed by Shin *et al.* [133], achieving an average precision and recall of 94.35% and 94.56%, respectively. Furthermore, classifiers implemented using Bayesian Network and Classification Via Clustering techniques achieved an average precision and recall in the range 13.52-13.64% and 69.86-70.18%, respectively, when used to identify files prone to vulnerabilities in a cross-project setting. The increased performance exhibited by the classifier strengthens the support for use of code complexity metrics as indicators of vulnerabilities and also highlights the importance of evaluating different classification techniques when attempting to implement a classifier.

In the studies summarized so far, a metric was empirically validated to indicate if a file was vulnerability prone. In theory, the metrics may be used to rank files by vulnerability proneness. Furthermore, with the knowledge of the direction of association between a metric and vulnerable files, one could apply the metric to identify changes that increases the vulnerability proneness of the file. For instance, since the metric SumEssential is known to be statistically significantly higher in vulnerable file than in neutral files [133], a change to a file `foo.c` that increases the value of SumEssential of `foo.c` above a certain predetermined threshold could be flagged for review. The study by Meneely *et al.* [85] performed empirical analysis at the level of changes (commits) to files to describe the characteristics of those commits that have historically contributed vulnerabilities to files. These commits were aptly called vulnerability-contributing commits or VCCs. In this study, the researchers proposed a novel variation to the traditional code churn metric, called interactive churn, which is a metric to quantify the extent to which developers are modifying source code written by other developers. The data set used in the study comprised of 124 VCCs identified by tracing 68 vulnerabilities in Apache HTTP to the commits that contributed to the vulnerability. The researchers found that (1) VCCs had higher churn and number of distinct authors, (2) took

an average of 1,175 days for a vulnerability contributed by a VCC to be resolved, (3) 13.53% of VCCs were present since the initial import of the project, and (4) 26.6% of VCCs were to files that had prior vulnerability fixes. The study demonstrates a potential for the use of metrics in continually assessing commits to a file with the intention of flagging those commits that increase the risk of file being vulnerability prone for review.

Traditionally, the metrics proposed in the vulnerability discovery literature summarized so far are derived from the product or process. In an empirical study, Scandariato [127], rather than proposing additional metrics, applied text mining to represent source code using frequencies of terms contained. The researchers hypothesized that a classifier build with the source code terms as features would be effective at identifying components likely to contain vulnerabilities. In an exploratory analysis of 20 Android applications, the researchers trained and tested two types of classifiers: Naïve Bayes and Random Forest. While the classifiers exhibited impressive performance metrics with precision ranging from 0.59 to 1.00 and recall ranging from 0.24 to 1.00, the response from the classifiers was not if a file is vulnerable or not but if the file has a static analysis warning or not. When the classifiers were trained using documented vulnerabilities in Drupal, a popular content management system written in PHP, the performance of the classifiers reduced to an average precision and recall of 0.57 and 0.77, respectively. Overall, the performance of classifiers built with text mining features was better than classifiers built with traditional metrics. In a collaborative study Walden *et al.* [149], compared software metrics and text mining features and found text mining features to be more effective at classifying files as vulnerable or neutral. The effectiveness of the text mining features is impressive, however, interpreting the features may not be straightforward. Even if interpretation was possible the insights are unlikely to yield actionable intelligence on the state of security of software. For instance, if the interpretation indicated that a change containing higher frequency of the keyword `if` is likely to be vulnerable, one cannot simply reduce the frequency of `if` to mitigate the vulnerability.

In a study similar to that conducted by Meneely *et al.* [85], Perl *et al.* [115] traced 718 vulnerabilities across 66 C and C++ projects to 640 VCCs. In the empirical analysis, the researchers compared 16 metrics collected from VCCs and other (possibly non-VCC) commits. The association analysis revealed that almost all the metrics were statistically significantly different in the sample of VCCs when compared to other commits. The researchers used historical vulnerability data to train and test a Support Vector Machine classifier and found that their classifier outperformed a tool used to identify security weaknesses called FlawFinder.

In a recent replication of prior work by Zimmermann *et al.* [160], Morrison *et al.* [91] collected and analyzed 29 metrics broadly categorized as churn, complexity, dependency, legacy, size, and pre-release vulnerabilities. While replication of prior work was successfully demonstrated with outcomes being consistent with that from prior work, the primary objective of the study was to understand the limitations of vulnerability prediction models that inhibit their adoption in the software development industry. In replicating prior work, researchers made some notable observations that could be the inhibiting factors. The granularity at which the prediction models operate is important because a model that predicts individual files as vulnerable is more useful than a model that predicts binaries as vulnerable as a single binary may contain source code from several thousand files. Since vulnerabilities are such a rare occurrence, the data sets used to train the prediction models are severely imbalanced with the neutral files being overwhelmingly high in number over vulnerable files. The challenge with imbalanced data sets is compounded when the granularity at which the prediction model must operate at is lowered. An approach to alleviate the granularity limitation could be to use machine learning techniques that are tailored specifically for learning

in situations when imbalance in data set is the norm. In the study, the researchers observed that Random Forest classifiers performed better than classifiers implemented using five other machine learning techniques.

The role of vulnerability discovery metrics is to assist developers in improving the security of software. In a recent study by Younis *et al.* [156], prior vulnerability discovery metrics were used to understand the differences in characteristics of vulnerable functions that have an existing exploit and those that do not have an existing exploit. The study was motivated by a need to understand the reasons for only small portion of disclosed vulnerabilities to have an exploit. The researchers examined 183 vulnerabilities across the Linux Kernel and Apache HTTP projects. For each vulnerability, the researchers identified the function that was fixed and collected eight metrics to quantify size, structure, ease of access, and communication of each such function. In the association analysis, the researchers found that vulnerable functions that had existing exploits were statistically significantly likely to have had fewer lines of code, decision paths and other functions that invoked them than vulnerable functions that did not have existing exploits. The researchers also evaluated the effectiveness of a set of classifiers in classifying vulnerable functions as (likely) to have an exploit or otherwise. The classifiers were built using subsets of metrics chosen via different feature selection approaches. A classifier implemented using Random Forest performed the best with an average precision and recall of 83% and 84%, respectively. If we had a classifier capable of identifying vulnerable functions with 100% precision and recall, the classifier proposed in this study could be used to further identify if the vulnerable function is likely to have an exploit. While the utility of the classifier may be a stretch to imagine, the metrics used in the models are still valuable.

In our search for prior literature related to vulnerability discovery we found some studies [141, 29, 13] in which the empirical analysis was conducted using historical vulnerability data obtained from sources other than the National Vulnerability Database (NVD) [1]. In a study of web applications, Smith and Williams [141] used number of hotspots (defined as a location in the source code where SQL statements are executed with input which has not been validated). In the empirical analysis, 158 security issues across 15 releases of two open source PHP web applications (WordPress and WikkaWiki) were used as proxy for vulnerabilities. The researchers found that (1) files with more hotspots were more likely to have other types of web application vulnerability, (2) files with more hotspots in one release were more likely to be vulnerable in the next release, and (3) more lines were changed to resolve security issues were hotspots. In a similar study of PHP web applications, Doyle and Walden [29] empirically analyzed three metrics in 14 open source PHP web applications. The analysis, however, used warnings from a static analyzer to indicate vulnerabilities rather than reported vulnerabilities. The researchers found that cyclomatic complexity was strongly correlated with number of static analysis warnings in a file. Bosu *et al.* [13] used a keyword search based approach, supplemented by thorough manual evaluation, to identify code reviews associated with VCCs. The researchers identified 413 VCCs in 10 open source projects. The objective of the study was to describe characteristics of the VCCs and code reviews associated with them. The researchers found that (1) the types of vulnerabilities identified by peer code review are diverse and represented all top ten security vulnerabilities ranked by the NVD, (2) code reviews typically find vulnerabilities that are easy to fix, (3) probability of a file containing a vulnerability increased with increase in churn, (4) modified files were more likely to contain vulnerabilities than new files, (5) authors of most VCCs were experienced developers but less experienced developers were 1.8 to 24 times more likely to author VCCs, and (6) developers affiliated with the organization sponsoring an open source project were more likely to author VCCs.

In summary, there is considerable scientific knowledge on vulnerability discovery in prior literature, however, as can be gathered from the summary of most relevant literature above, the knowledge is scattered. Furthermore, there is rich diversity in the approaches used to validate the vulnerability discovery metrics. We hope that the systematizing the knowledge will help the transition of knowledge into practice.

## 3.2 Feedback in Software Engineering

In a tool-assisted approach to software development today, developers receive feedback from a variety of tools such as integrated development environments that aggregate errors and warnings from compiling, building, testing, and/or statically analyzing the source code. In addition to tools, practices such as code review create an opportunity to receive feedback from other developers. There are many studies in prior literature in which feedback during software engineering, generated automatically or provided by humans, was evaluated to characterize effective feedback. We can leverage the recommendations from these prior studies to provide feedback on security in a way that it useful to developers. We summarize relevant studies on feedback from prior literature here.

In the year 2007, Layman *et al.* [62] conducted a study to understand the factors that influence developers to address faults detected by automatic fault detection tools. The researchers conducted a controlled study of the experience of 18 developers using a custom automatic fault detection platform called AWARE [142]. AWARE is a plug-in for the Eclipse IDE that continuously provides developers a listing of faults prioritized by severity. The researchers examined transcribed responses from the study participants to identify seven categories of themes that emerged from the responses. In interpreting the themes, the researchers made concrete recommendations that could encourage developers to use automatic fault detection tools. Some of the recommendations were (1) for the fault descriptions to be informative and relevant, (2) to have the severity assigned by the tool to match developers' perceived severity of the fault, (3) allow customization of when the faults were shown, and (4) to have fewer false positives. Even after ten years, these recommendations are still relevant, as we observed in reviewing more recent studies on automatic feedback generation.

Johnson *et al.* [52] conducted a study similar to that conducted by Layman *et al.* [62] but to identify reasons for developers to *not use* static analysis tools. FindBugs[1] was the static analysis tool used in the study. The researchers used semi-structured interview to capture the experience of 20 developers (16 professional developers and 4 graduate students with prior professional development experience) using FindBugs to find and resolve issues in a piece of software. Using open coding of the responses from interviews, six themes were identified. In five of the six themes, the responses were overwhelmingly negative. The researchers highlighted several reasons for static analysis tools to be underutilized in practice. Some of the most notable limitations were (1) poorly presented results with false positives outweighing true negatives, (2) loss of context when scrolling through a long list of warnings, (3) lack of support for collaboration, (4) lack of support for justified suppression of warnings, (5) lack of integration with workflow, and (6) interruption to developers' thought process.

In a similar study, Foss and Murphy [34] evaluated if showing developers code stability in the source code editor affect their behavior. The researchers developed ChangeMarkup, an IntelliJ IDE plug-in, to show stability warning at the line level using colored markers. The plug-in showed two types of stability warnings: one used the age of the source code line as a metric and the other used change size (churn) as a metric. Five developers participated

[1] `http://findbugs.sourceforge.net/`

in the study for two weeks. The participants were provided a questionnaire four times through the study to capture the change (if any) in the behavior of the developers. After the study had concluded, each participant was interviewed for thirty minutes to understand the experience of the participants. The researchers found that (1) developers preferred even simpler cues than those provided by ChangeMarkup, (2) developers were drawn by unfamiliar visual cues in the editor but they seldom used features to drill down a warning to get more information, (3) and developers tend to ignore warnings when they become predictable. The observations from this study provide insights into developers' preferences for receiving feedback. The researchers also suggest that code stability warnings are better suited to be shown during other processes such as code reviews.

Building on work by Layman *et al.* [62], Foss and Murphy [34], and Johnson *et al.* [52], Lewis *et al.* [63] and Sadowski *et al.* [126] conducted large-scale studies involving professional developers at Google. The study by Lewis *et al.* [63] was exploratory with a goal to understand if the use of bug prediction affects developers' behavior in code reviews. In the first phase of the study, developers' perception of the effectiveness of three state-of-the-art bug prediction algorithms was assessed. 19 developers from the Google Engineering Tools department volunteered to participate in the study. Each developer was provided with three lists of 20 files from two projects determined to be bug prone by each of the three bug prediction algorithms. The task was to classify the files as bug-prone or not-bug-prone and to rank the lists by extent of bug proneness of files. The aggregation of the responses from the task revealed that developers seemed to prefer the outcome from one of the three bug prediction algorithms. After the first phase, researchers had several informal discussions with the participants of the study. The informal discussions revealed several common themes among developers' requirements from bug prediction and reporting systems. Some of the notable themes were (1) actionable warning messages with obvious reasoning, (2) biased toward newer files, and (3) effectiveness and scalability. In the second phase of the study, researchers modified the most effective bug prediction algorithm identified in the previous phase to bias the prediction toward newer files. The modified bug prediction algorithm was integrated with the code review system such that a file determined to be bug prone would be flagged for review. The objective in the second phase was to understand if the presence of the flag affects developers' behavior in the review. The average time for the approval of and average number of comments in review containing bug-prone files was compared for three months before and after the deployment of the integration. In statistical analysis, researchers found no difference between the two samples of metrics leading to the conclusion that the introduction of bug prediction was not likely to change developers' behavior during code review. The lack of an opt-out feature, confusion between authors and reviewers on the fix, files with technical debt being repeatedly flagged, and auto-generated files being flagged were some of the reasons that developers mentioned to the integration to be perceived as ineffective. Despite the limitations of the bug prediction algorithms, developers at Microsoft use them to improve the quality of their software [91].

As suggested by Lewis *et al.* [63], improving the effectiveness of bug prediction is certainly desirable, however, seamlessly integrating existing algorithms with developer workflow has its benefits as well. The study by Sadowski *et al.* [126] attempted to accomplish just that goal. Sadowski *et al.* [126] developed a program analysis platform, called Tricorder, which integrated with a code review system (similar to Gerrit[2]) used internally at Google. The study contains a description of the experience of developing and deploying Tricorder at a scale that spanned the entire source code repository at Google. Tricorder was designed to follow the microservices architecture that simplifies scalability, extensibility, and

[2] `https://www.gerritcodereview.com/`

reliability. The TRICORDER architecture is similar that of a platform, called Mediam, proposed by Beschastnikh *et al.* [10] which is aimed at accelerating the adoption of software engineering research in the industry. The analyzers in TRICORDER are analogous to the bots in Mediam. TRICORDER, when notified of a change submitted for review by a developer, would launch one or more analyzers to examine the change for issues. The analyzers would run independent of one another and report issues back to TRICORDER which would then aggregate the issues and present them as comments in the code review system. Each comment posted by TRICORDER was accompanied by four links: (1) NOT USEFUL, used to provide feedback to the developer of the analyzer, (2) PLEASE FIX, used, by a reviewer, to indicate to the author that the comment must be addressed, (3) PREVIEW FIX, used, by the author, to preview automatically generated fix for the issue described in the comment, and (4) APPLY FIX, used, by the author, to apply the automatically generated fix. The effectiveness of TRICORDER, and its analyzers, was evaluated using the number of clicks on each of the four links. TRICORDER was introduced to Google developers in July, 2013. Using the data collected from TRICORDER, the researchers demonstrated the overwhelming utility of such a platform. Using the feedback received from developers, the researchers made recommendations to developers of analyzers, the most notable of which were (1) easy to understand messages, (2) fewer false positives, (3) scalable, and (4) tight feedback loop between users and developers of the analyzers.

In addition to the research studies summarized so far, we found a U.S. Patent application by Allen *et al.* [4] describing a method, system, and implementation approach for providing real-time feedback to developers on their most common programming mistakes. We will follow a similar approach to provide developers with feedback but with some key differences: (1) feedback will provided asynchronously when a developer submits a change for review and (2) feedback will be provided not only to the developer implementing the change but also to the developer(s) who may be reviewing the change.

Despite concerns that software development supported by bots could disrupt developer productivity [143], using bots seems to be an elegant approach to accelerate the adoption of software engineering research in the industry [10]. We will use the recommendations from these studies from prior literature to inform the decisions we make in developing our approach to providing automatically generated feedback on security.

## 3.3 Summary

As noted in summarizing the literature on feedback in software engineering, there are several opportunities for providing feedback to developers in a way that it prompts a conversation. The TRICORDER platform described by Sadowski *et al.* [126] is a prime example of leveraging the code review system to provide warnings from static analyzers in a way that it implicitly enforces peer accountability.

Static analysis has helped developers improve the overall quality of software by uncovering defects [108]. Unfortunately, vulnerabilities are not the same as defects [104, 93]. Traditional approaches to static analysis tend to use pattern matching to find code-level and technology-level problems. Leveraging static analysis to uncover vulnerabilities may be non-trivial as patterns matching yet undiscovered vulnerabilities may not exist [19]. Furthermore, vulnerabilities that become apparent only when the source code is executed under very specific circumstances cannot be statically identified. Fortunately, researchers have proposed a plethora of metrics that may be useful in discovering vulnerabilities as software is engineered. The literature on vulnerability discovery summarized above provides a sense of

the wealth of available knowledge.

The goal in our research is to leverage the best of both areas of research to support developers in engineering secure software. We wish to leverage effective vulnerability discovery metrics to provide automatically generated feedback on security in ways that have been shown to be effective in the industry.

# Chapter 4

# Beyond the Attack Surface

When reasoning about software security, researchers and practitioners use the phrase "attack surface" as a metaphor for risk. Enumerate and minimize the ways attackers can break in then risk is reduced and the system is better protected, the metaphor says. But software systems are much more complicated than their surfaces. We propose function- and file-level attack surface metrics—proximity and risky walk—that enable fine-grained risk assessment. Our risky walk metric is highly configurable: we use PageRank on a probability-weighted call graph to simulate attacker behavior of finding or exploiting a vulnerability. We provide evidence-based guidance for deploying these metrics, including an extensive parameter tuning study. We conducted an empirical study on two large open source projects, FFmpeg and Wireshark, to investigate the potential correlation between our metrics and historical post-release vulnerabilities. We found our metrics to be statistically significantly associated with vulnerable functions/files with a small-to-large Cohen's $d$ effect size. Our prediction model achieved an increase of 36% (in FFmpeg) and 27% (in Wireshark) in the average value of $F^2$-measure over a base model built with SLOC and coupling metrics. Our prediction model outperformed comparable models from prior literature with notable improvements: 58% reduction in false negative rate, 81% reduction in false positive rate, and 548% increase in $F^2$-measure. These metrics advance vulnerability prevention by(a) being flexible in terms of granularity, (a) performing better than vulnerability prediction literature, and (a) being tunable so that practitioners can tailor the metrics to their products and better assess security risk.

The study presented in this Chapter is published in the Proceedings of the 2016 ACM Workshop on Software PROtection (SPRO) [100].

## 4.1 Motivation

The *attack surface* metaphor is often invoked as a way to assess the security risk of large systems [45, 66, 47, 72, 73]. The metaphor goes like this: if an attacker has more avenues to

enter the system, then the system is at a higher risk. Software systems provide these avenues in their inputs and outputs, where inputs can take in potential exploits and outputs provide information on how data is processed (e.g. lack of sanitization). Thus, broadly speaking, developers can understand their security risks by understanding their inputs and outputs. This metaphor has been the inspiration for security risk metrics such as the number of *entry points* and *exit points* [72, 74, 71] and is even in practice at Microsoft [66].

Historically, attack surface metrics have focused on the system's "perimeter" and hence fail to capture what happens beyond the entry and exit points. A simple source code change, such as the addition of an API call deep within the system, can have a drastic effect on security risk because large software systems are vastly interconnected and the API call could be connecting two subsystems that were earlier disconnected. If the attack surface metaphor is about examining avenues of attack, then risk analysis should be about how attacker *traverses* the software system.

The *call graph* [43, 42, 3, 31] can be used as a basis for a model of attacker behavior because it contains the overall system structure. Random walks through a call graph can serve as an approximation for attacker searching (manually or automatically) for vulnerabilities. With network analysis techniques [15, 83, 116, 81] such as centrality, random walks, and geodesic paths, we can provide developers with fine-grained risk metrics at the function- or file-level that can be used to continually assess the impact of a source code change on security risk.

*The goal of this research is to assess security risk through an empirical understanding of the relationship between vulnerabilities, individual functions or files, and the attack surface of a software system.* We apply the attack surface metaphor described in prior literature [45, 47, 71] to the call graph and propose function- and file-level metrics—proximity and risky walk—that simulate random walks across functions/files to account for system structure beyond the surface. We empirically analyze our metrics in two large open source projects: the FFmpeg media transcoder and the Wireshark network protocol analyzer to understand if the metrics are associated with historical vulnerabilities. We also investigate the sensitivity of the metrics to call graph collection approach (static-only vs. static+dynamic), as well as parameter tuning for our random walk metric.

We address the following research questions:

**RQ 1 - Association**

Is a function/file more likely to be fixed for a post-release vulnerability if:

(a) it is near the attack surface or dangerous points?

(b) it has a higher probability of being traversed on a random walk from the attack surface?

**RQ 2 - Prediction (Base)**

Do proximity and risky walk metrics improve the performance of a base prediction model built with SLOC and coupling metrics?

**RQ 3 - Prediction (Prior)**

How do the prediction models built with proximity and risky walk metrics compare with prior vulnerability prediction literature?

The research contributions of this work are:

- A method for applying the attack surface metaphor to individual functions/files so that developers can quantitatively assess security risk;
- Implementation of our method as an open source project [99];
- Empirical evaluation of our method in two large open source projects.

## 4.2 Metric Motivation

In this section we discuss some key decisions that led to the formation of our risk metrics. Modeling attacker behavior using graphs is not new [132, 79], nor is using call graphs [72, 74, 154], but our approach of combining attack surface and call graph is unique to our knowledge.

Historically [45], software attack surface researchers have concluded that reducing points of entry/exit should reduce the number exploits attackers can use. We expand the scope of that argument by applying the principle of *defense in depth*. We wish to consider risks of attack by aggregating how the system is interconnected, still using the attack surface as a starting point, to model attacker behavior.

Consider the attack surface metaphor as it applies to medieval castles. If a castle has many different entrances and exits, attackers will also have many different ways of invading the castle. Reduce entrances and the security will improve. But, an inner courtyard might also be particularly risky because it is the point of convergence for multiple pathways into the castle. A renovation within the castle might change attacker behavior without changing the external entry and exit points. By combining both structure and surface, we can better understand our system from the inside out.

**Why use call graphs?**

Call graphs provide an inexpensive, automated way of measuring how the system is interconnected. Most programming languages support call graph collection, so these metrics could be easily adopted into, say, a continuous integration build. Call graphs are also the next natural step from the attack surface methods because entry/exit points in prior attack surface literature [72, 74, 71] are actual methods.

While call graphs are simple to collect they are, in practice, imperfect representations of what can happen at runtime because of pointer manipulations. In our approach discussed in the subsequent section, we discuss how one could mitigate this concern and how, in our empirical study, we found our call graph to be a close representative of potential attacker pathways. We also examined the sensitivity of static-only vs. static+dynamic call graph collection in our empirical study in Section 4.4.2.

Once customized to the build process of a software system, our proposed approach is entirely automated. We envision developers using these fine-grained metrics as an informative part of their everyday development workflow to prioritize their software protection efforts such as code reviews and penetration testing.

**Why use random walk metrics?**

Many network analysis applications rely upon *geodesic* path (i.e. "shortest path") metrics to provide analysis. Geodesic path metrics are useful in social situations where distance is to be measured and the attempt to find a shortest path (e.g. two humans who are "friends" are also "friends-of-friends", but the shortest path makes the most sense in that domain). We use geodesic paths whenever we want to gauge potential distance (e.g. "distance to the attack surface").

Computers, however, do not execute methods based on shortest paths, they execute methods based on inputs. For attackers searching for a vulnerability, perhaps via a fuzz testing tool or through manual experimentation, traversals of the network would look more random. An attacker, via his inputs, would execute commonly-used methods, leading to a higher risk of attack if those methods had a vulnerability. Thus, we use random walk metrics to simulate the attacker behavior of exploring the system.

**Why use PageRank and not regular Random Walk?**

The standard Random Walk metric is useful when discussing how someone traverses a network infinitely (e.g. car traffic patterns). Attackers, however, are constantly starting over their traversals–at the attack surface. The PageRank metric, based on Google's algorithm of modeling web surfing behavior, is a Random Walk with additional parameters of(a) damping factor (a probability of "starting over"), and (b) personalization vector (the probability of starting at a particular method). The concept of constantly starting over at the attack surface and then conducting a random walk fits with the scenario of an attacker exploring for a vulnerability. Thus, to incorporate the notion of attack surface into call graph centrality, we found PageRank to be the most appropriate metric.

## 4.3 Proposed Metrics

In this section, we introduce the proximity and risky walk metrics in the context of the attack surface metaphor introduced earlier. A unique (read desirable) feature of our metrics is that they can be defined at either function- or file-level, enabling developers to choose the level of granularity in risk assessment.

The high-level approach to collecting the metrics, at either the function- or file-level, may be summarized as follows:

Step 1: Obtain the call graph

Step 2: Identify Entry, Exit, and Dangerous Points

Step 3a: Compute Proximity metrics

Step 3b: Compute Risky Walk metric

We have developed an open source tool, called *Attack Surface Meter* [99], that enables the collection of the metrics proposed in this chapter for a software system written in the C programming language. As of this writing, *attack surface meter* is capable of parsing call graphs obtained from GNU cflow and GNU gprof. The *attack surface meter* may be extended to measure software systems written in other programming languages by defining a parser for the call graph generated by an appropriate language-specific utility.

### Step 1: Obtain the call graph

The proximity and risky walk metrics are defined on the call graph representation of a software system. The call graph represents a series of steps that an attacker effectively takes when attempting to exploit a vulnerability. In our definition, a call graph is a collection of directed relationships from *caller* functions to *callee* functions, represented as a symmetric directed graph. That is, if an attacker's exploit accesses a callee, it can potentially access the caller, and vice versa. We use a symmetric directed graph instead of an undirected graph so that we can weight returns differently than calls. (Note: our predecessors [72, 74, 154]

have used directed but not symmetric call graphs, which we believe does not fully account for function returns.)

The caller—callee relationships may be deduced in two ways:(1) *static analysis*, where the source code is parsed and analyzed, or (2) *dynamic analysis*, where the software system is profiled during execution. Both techniques are imperfect: static analysis has limited support for language features such as function pointers and polymorphism, and dynamic analysis requires software to compile and all possible execution paths to be exercised. Thus, the call graph is always an approximation as Grove et al. mention when describing *soundness* of call graphs [42]. We hypothesize that the level of approximation achieved by the call graph may be improved if static and dynamic analyses are used in unison. In Section 4.4.2, we present the sensitivity of our metrics to static-only vs. static+dynamic analysis.

Developers familiar with their own systems can manually inspect the call graph for soundness, as we did in both of our historical studies. Based on applying our technique to historical case studies, we also used two heuristics for gauging if the collected call graph has a representative set of edges. The two heuristics we used were:

**Number of Fragments ($f$)** The number of strongly connected components in the call graph

**Monolithicity ($m$)** The percentage of total functions that is in the largest strongly connected component.

We note that these heuristics are useful in systems that are "monolithic", that is, systems that are intended to compile into one massive call graph. While this happened to be true in our empirical studies, it may not be in, say, an API with intentionally disconnected subsystems such as `glibc`.

In practice, software systems often have functions that are never invoked or functions that are invoked only when testing. These functions will appear as islands in the call graph, so the ideal $m$ is not necessarily 100% and the ideal $f$ is not necessarily one. The ideal value of $m$ and $f$ vary between systems, and must be taken into consideration.

The call graph described so far is conventional in that the edges in the graph represent function call/return. The edges in the conventional call graph may be used to deduce call/return relationships between files as well. The *file-level call graph* enables the proximity and the risky walk metrics to be collected at a file-level, providing an alternate approach to security risk assessment.

### Step 2: Identify Entry, Exit, and Dangerous Points

Attackers need places to send their attacks, or places where they might start the reconnaissance for their attacks. From Manadhata et al. [71], we defined these functions as *entry points* (where data enters in) and *exit points* (where data exits out). Additionally, attackers may be more likely to target functions that make system calls deemed dangerous. We call these functions as *dangerous points*. There may be other criteria in which a function could be deemed dangerous (e.g. function handling sensitive information specific to an application), however, we restrict ourselves to only functions that make dangerous system calls.

The language used in the development of a system and its operating environment determine how inputs, outputs, and the dangerous system calls are identified. For instance, in the context of a web application, an entry point could be a method that saves form-posted data to a database, whereas, an exit point could be a method that formats data for a web

page. In the context of a C program: any function invoking a C standard input function (e.g. `scanf`, `getc`, etc.) is an entry point, whereas, any function invoking a C standard output function (e.g. `printf`, `putc`, etc.) is an exit point. Again, in the context of a C program, any function making a dangerous system call (e.g. `chown`, `fork`, etc.) is a dangerous point. Defining entry and exit points for all available technologies may be an open problem, but the most popular set of functions for the C standard library is in Appendix A of Manadhata and Wing [73]. Similarly, the set of available system calls is dependent on the version of the C standard library used during the development of a software system. We have used the system calls with threat level 1 through 3 enumerated by Bernaschi et al. [9].

The notion of entry, exit, and dangerous points may be extended to the file-level by applying the following heuristic: a file is an entry point, exit point, or dangerous point if it contains at least one function that is an entry point, exit point, or dangerous point, respectively.

### Step 3a: Compute Proximity Metrics

As an attacker's exploit enters the system, functions/files that are near entry and exit points are likely to be involved with handling user data. In the call graph, nearby ancestors (i.e. functions/files that can reach a given function/file) may be more likely to have security risks, so we use an unweighted shortest path algorithm to determine the distance from a given function/file to the attack surface. Since functions/files may be reachable from multiple entry and exit points, we average the shortest path lengths. We chose to use the average of the shortest path lengths to better approximate the reality of function/file invocation pattern. In addition to measuring the distance to the attack surface, we also measure the distance of a function/file to dangerous points that were defined in the previous step.

Our three proximity metrics (i.e. proximity to entry, proximity to exit, and proximity to dangerous) are defined as:

**Definition 1** *Proximity of a function/file* `foo` *to entry points, exit points, or dangerous points is the mean of the shortest unweighted path lengths to all functions/files reachable from* `foo` *that are entry points, exit points, or dangerous points, respectively.*

The power of the proximity metric is in its sensitivity to the (unforeseen) ripple effect of a source code change on functions that may have not been directly modified by the developer. For example, suppose a developer working on a function, `readMessage`, adds a call to `pingServer`. The proximity of `readMessage`, and all its descendants (i.e. functions reachable from `readMessage`), to the entry surface would decrease if `pingServer` is near an entry point. Conversely, a refactoring effort on `pingServer` that separates concerns of input validation or secure memory management would increase the proximity of `readMessage` to the entry surface without a direct change to it.

### Step 3b: Compute Risky Walk Metric

Attackers looking for vulnerabilities may have limited knowledge of the system's source code and are essentially exercising different execution paths in the system hoping to find a vulnerability. This behavior of the attacker is similar to that of a World Wide Web user ("surfer") searching for a piece of information. The surfer starts at, say, the results from a search engine and follows a series of links until she finds the information she was looking for. Or perhaps she deems the search futile, at which point she returns back to the starting point

and follows a different series of links. The starting point of the surfer is analogous to the entry points of the software system and the act of following a link is analogous to invoking a function in the software system. However, the attacker has no direct control over the series of function calls that the system makes in response to a particular input. As a result, the attacker resorts to trying several entry points with varying inputs.

In addition to ranking web pages [113], twitter users [58], and improving recommender systems [76], PageRank algorithm has found application in the realm of security as well [147, 79].

The PageRank algorithm uses three configurable parameters in addition to the call graph, they are:

- a *personalization vector*, $v$, that contains the probability that a random walk starts at a given node,
- a *damping factor*, $\alpha$, that defines the probability that an attacker will continue the random walk across the call graph without abandoning the current walk and starting over, and
- an *edge weights vector*, $w$, that contains the edge weights used to derive the probability that a random walk traverses one of the many possible edges from a given node.

Wills [153] presents an elaborate description of the mathematics behind the computation of the page rank and the role of these parameters in the algorithm.

In the context of security risk assessment, these parameters must be chosen with the intent of reflecting attacker mindset. For instance, we may want to assign a higher weight to edges terminating at functions that were fixed for vulnerabilities in the past to model the likelihood that an attacker may try to attack past vulnerable functions hoping to uncover a new vulnerability. Similarly, some software systems may have defenses in place to wraparound standard library functions known to be used incorrectly by developers. In such cases, we may want to assign a lower weight to edges terminating at such defensive functions.

In the empirical analysis of the risky walk metric in FFmpeg and Wireshark, we carried out an extensive parameter tuning exercise (detailed in Appendix B) to identify a robust set of parameters. Users of our approach may choose to use the parameters we arrived at in our study as our parameters ended up being similar across case studies. Alternatively, users may choose to use our weights as a starting point and adapt them to their own software systems.

Our Risky Walk metric is defined as follows:

**Definition 2** *Risky walk of a function/file is the PageRank of that node in the call graph computed with a personalization vector, a damping factor, and an edge weights vector tuned to simulate attacker behavior.*

The power of risky walk metric is that it aims to simulate, by means of probabilities, the behavior of a typical attacker, specifically during the reconnaissance phase of an attack. The risky walk of a function/file is the probability that a random execution of the system, with inputs tailored to uncover vulnerabilities, will result in the function/file being invoked. In practice, risky walk of a function/file may be extremely small in a system with large number of functions/files, so viewing the logarithm of the risky walk or simple a ranking can make the values easier to interpret.

### 4.3.1 An Example

Shown in Figure 4.1 is the call graph of a sample C program with the attack surface highlighted with dotted ellipse. The nodes represent the functions in the program. The solid directed edge represents call to a function, whereas, the dotted directed edge represents return from a function. The function `read_config` is an entry point because it calls an input function `scanf`. The functions `compute_foo` and `pretty_print` are exit points because they call the output function `printf`. The entry and exit points are shaded gray. For simplicity, the graph does not show any dangerous points.

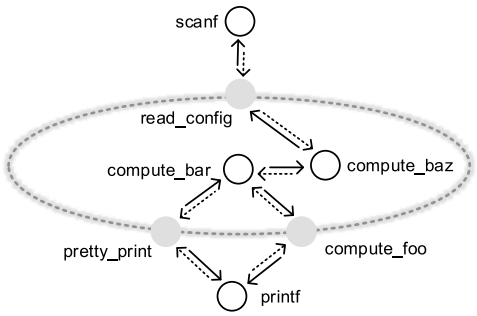



Figure 4.1: Attack surface visualization of a sample C program

The proximity to entry for `compute_baz` is 1 and for `compute_bar` it is 2. The proximity to exit for `compute_baz` is $(2+2)/2 = 2$ and for `compute_bar` it is $(1+1)/2 = 1$. Let $\alpha = 0.85$, vector $v$ contain 0.3125 for entry and exit points and 0.03125 for the other functions (i.e. attacker is 10 times more likely to start at entry or exit point), and the vector $w$ contain 10 for call edges and 5 for return edges. For risky walk, the function `compute_bar` would have a total weighting of $10 + 10 + 5 = 25$ for the outgoing edges. Each edge weight is computed as a proportion of the total, for example, the `compute_bar` to `compute_baz` edge probability would be $10/25 = 0.4\%$. The page rank of `compute_baz` will be 0.29 and `compute_bar` will be 0.27.

## 4.4 Methodology

In this section, we describe the methodology used in the empirical evaluation of the proximity and risky walk metrics in the context of two large open source projects: FFmpeg and Wireshark. At a high-level, the empirical evaluation was conducted in four phases, they are:

Phase I: Metric Collection

Phase II: Function/File Labeling

Phase III: Association Analysis

Phase IV: Regression Analysis

All statistical tests were executed on `R` version 3.2.3 [118].

### 4.4.1 Study Subjects

We chose two large open source projects as subjects of study in the empirical evaluation of our metrics. The motivation for choosing our subjects of study were:(a) large, popular, and open source projects, (b) well-kept vulnerability fix records, (c) substantial development history for tracking vulnerabilities over time, and (d) automated regression test suites to compare the sensitivity of static-only vs. static+dynamic analysis.

**FFmpeg** is a popular open source media transcoding library that is capable of encoding, decoding, multiplexing, demultiplexing, streaming, filtering, and playing an enormous variety of media. FFmpeg is used by other projects such as Google Chrome and the VLC Media Player. Since 2009, FFmpeg has had 19 major releases, 217 patch releases, and 237 vulnerabilities. In this study, we collected metrics for 16 of the 19 major releases of FFmpeg, mining 675 vulnerability-fixing commits to identify 280 unique functions that were fixed for a post-release vulnerability. On average, each major release of FFmpeg has 536k source-lines-of-code (SLOC) and 12,908 functions spread across an average of 1,865 files. In terms of SLOC, the average length of each file is 306 and that of each function is 30. FFmpeg has an extensive, automated regression test suite call FATE[1], which we leveraged for dynamic analysis.

**Wireshark** is an open source network protocol analyzer that has come to be the de facto standard for sniffing network data. The current release of Wireshark supports the analysis of 2,000 network protocols. Since 2008, Wireshark has had 8 major releases, 118 patch releases, and 312 vulnerabilities. In this study, we collected metrics for 7 of the 8 major releases of Wireshark, mining 590 vulnerability-fixing commits to identify 1,705 unique functions that were fixed for a post-release vulnerability. On average, each major release of Wireshark has 2,081 kSLOC and 53,350 functions spread across an average of 2,593 files. In terms of SLOC, the average length of each file is 855 and that of each function is 24. We collected dynamic analysis data by developing a simple test runner script to invoke the Wireshark GUI (and its command line variant, TShark) for a set of 1,877 packet capture files typically used for regression testing by the Wireshark team.

### 4.4.2 Phase I: Metric Collection

In this section, we apply the method introduced in Section 4.3 to collect the proximity and risky walk metrics from the releases of FFmpeg and Wireshark considered in our study. In addition to *attack surface meter*, we have developed another open source application, called *Attack Surface Evolution*,[2] to facilitate the automated (and parallel) collection of the metrics for multiple releases of a software system. The metrics collected are saved to a database for further analysis.

We note that we collected, and analyzed, the metrics at both function- and file-level to evaluate the utility of the metrics at different levels of granularity.

#### Step 1: Obtain the call graph

We chose two popular call graph generation utilities: GNU cflow,[3] a static call graph generation utility, and GNU gprof,[4] a dynamic profiling utility, to obtain the call graphs of

[1] `https://ffmpeg.org/fate.html` [2] `https://github.com/nuthanmunaiah/attack-surface-evolution`
[3] `http://www.gnu.org/software/cflow/` [4] `https://sourceware.org/binutils/docs/gprof/`

FFmpeg and Wireshark. We refer to these as `cflow` and `gprof` in the remainder of the section, respectively.

In order to determine the need for dynamic analysis, we use the heuristics—number of fragments and monolithicity—from Section 4.3. We obtain static and dynamic call graphs for the most recent release of FFmpeg and Wireshark considered in our study and use the heuristics to determine if there is a need for dynamic analysis or not. The most recent releases of FFmpeg and Wireshark considered in this study are 2.5.0 and 1.12.0, respectively. Since FFmpeg and Wireshark are intended to compile into a single system, we expect number of fragments to be low and monolithicity to be high. The number of fragments ($f$) and monolithicity ($m$) of the FFmpeg version 2.5.0 and Wireshark version 1.12.0 are given in the Table 4.1.

Table 4.1: Number of Fragments ($f$), Monolithicity ($m$), and mean value of Proximity to Entry ($p_{en}$), Proximity to Exit ($p_{ex}$), Proximity to Dangerous ($p_{da}$), and Risky Walk ($rw$) from static and static+dynamic analysis of FFmpeg version 2.5.0 and Wireshark version 1.12.0

| **Subject (Version)** | **Analysis** | $f$ | $m$ | $\mu_{p_{en}}$ | $\mu_{p_{ex}}$ | $\mu_{p_{da}}$ | $\mu_{rw}$ |
|---|---|---|---|---|---|---|---|
| FFmpeg (2.5.0) | Static | 182 | 0.979 | 3.669 | 3.807 | 3.498 | 6.530E-05 |
| | Static+Dynamic | 124 | 0.985 | 3.666 | 3.832 | 3.546 | 5.727E-05 |
| **Interpretation**: Appending call graphs obtained through dynamic analysis decreased the number of fragments and marginally increased the monolithicity. Furthermore, there was a non-trivial change in the mean value of risky walk metric. Hence, we use dynamic analysis when obtaining the FFmpeg call graph. | | | | | | | |
| Wireshark (1.12.0) | Static | 78 | 0.998 | 4.197 | 4.408 | 4.335 | 1.527E-05 |
| | Static+Dynamic | 82 | 0.998 | 4.195 | 4.403 | 4.334 | 1.525E-05 |
| **Interpretation**: Appending the call graphs obtained through dynamic analysis neither decreased the number of fragments nor increased the monolithicity. Furthermore, there was a trivial change in the mean value of the metrics. Hence, we do not use dynamic analysis when obtaining the Wireshark call graph. | | | | | | | |

`cflow` and `gprof` call graphs are saved to the disk as plain-text files. The *attack surface meter* parses the textual call graph files and produces a single call graph that represents the software system. The *attack surface meter* uses NetworkX[5] version 1.9.1 to represent the call graph. The nodes in the call graph represent the functions in the software system and the edges represent transfer of control.

#### Step 2: Identify Entry, Exit, and Dangerous Points

The C standard input and output functions, used to identify the entry points and exit points in FFmpeg and Wireshark, are the same as those listed in Appendix A of previous work [73] by Manadhata. The dangerous system calls used in our study are the system calls with threat level 1 through 3 enumerated by Bernaschi et al [9].

With the entry, exit, and dangerous points identified, functions belonging to the C standard library were removed from the call graph to prevent these functions from being accounted for in the computation of our metrics.

#### Step 3a and 3b: Compute Proximity and Risky Walk Metrics

The *attack surface meter* has methods to compute the proximity and risky walk metrics for a given function or file. These methods use the `shortest_path_length` and `page_rank` methods from the NetworkX API.

### 4.4.3 Phase II: Function/File Labeling

To evaluate the efficacy of our metrics as an indicator of vulnerabilities, we must understand *fixes* to historical post-release vulnerabilities and identify those functions/files that were vulnerable in the past. We labeled functions/files that were fixed for post-release vulnerability as *vulnerable* and all other functions/files as *neutral*. We refer to these two groups of functions/files as vulnerable functions/files and neutral functions/files, respectively.

We begin by collecting a list of publicly disclosed vulnerabilities from Common Vulnerabilities and Exposures (CVE)[6], National Vulnerability Database (NVD)[7], or the security advisories section on the project's website. The security advisories section on the project's website is more suited for our purposes as it contains the versions of software affected by a vulnerability and information that helps trace vulnerability fixes to the source code. Furthermore, the security advisories are released only when a publicly-disclosed vulnerability is acknowledged and fixed by the project team. The FFmpeg and Wireshark project teams post their security advisories at `https://www.ffmpeg.org/security.html` and `https://www.wireshark.org/security/`, respectively. In our study of FFmpeg and Wireshark, for each historical vulnerability fixed by the development team, we collected the commit identifier that the project team reports as containing the fix. For each vulnerability-fixing commit identifier collected, we generated a patch using the `git` client and parsed the patch output (similar to [86]) to identify the name of the functions/files affected by the commit. We manually examined a random sample of the patches to ensure that the fix commit was not combined with other changes. In our sample, we found none of these cases to be true for FFmpeg and Wireshark.

[5] `http://networkx.github.io/` [6] `https://cve.mitre.org/` [7] `https://nvd.nist.gov/`

### 4.4.4 Phase III: Association Analysis

We used the non-parametric Mann-Whitney-Wilcoxon (MWW) test to understand how well the proximity and risky walk metrics reflect the reality of historical post-release vulnerabilities. We consider the association between a given metric and post-release vulnerabilities to be statistically significant if the p-value is less than 0.05. We use the population median to determine if a metric is higher (or lower) for vulnerable functions/files when compared with neutral functions/files.

Association analysis merely reveals if there is a statistically significant difference between the distribution of the metric values collected from a population of vulnerable functions/files from that of the metric values collected from a population of neutral functions/files. We complemented the association analysis with Cohen's $d$ effect size evaluation to assess the *strength* of association (if any, as revealed by MWW test). We used the heuristics proposed in Cohen's $d$ literature [24] when interpreting the effect size. According to the heuristic, an effect is considered large if $|d| \geq 0.8$, medium if $|d| \geq 0.5$, small if $|d| \geq 0.2$, negligible otherwise.

### 4.4.5 Phase IV: Regression Analysis

In the regression analysis phase, the goal is to assess if the proximity and risky walk metrics can be used in building a regression model capable of predicting the likelihood of a function/file needing a fix for a post-release vulnerability in the future. A necessary condition in the evaluation of the efficacy of such a model is to assess if it performs better than a base prediction model built with SLOC and coupling metrics. The coupling metrics used in this study are the structural variant of fan in and fan out of a function/file. *Fan in* is the number of functions/files that call a given function/file and *fan out* is the number of functions/files that a given function/file calls. We chose to use SLOC, fan in, and fan out in building the base model because these metrics have been shown to be good predictors of vulnerabilities [160, 133].

While fan in and fan out were collected directly from the call graph, SLOC was measured using Scitools Understand[8]. Functions/files for which SLOC was not available from Understand were omitted when training and testing the model. We note that SLOC was available for all functions/files that were fixed for a post-release vulnerability.

We used two approaches in the training and testing of the regression models:(a) Cross–validation and (b) Next release validation. We used precision, recall, and $F^2$-measure to evaluate the performance of a model against the base model. In contrast to $F^1$-measure, the $F^2$-measure weights recall higher than precision. A model that exhibits a higher recall is desirable in vulnerability prediction [87, 22, 133]. We note that the performance of a model was evaluated if and only if the model had at least one statistically significant (p-value $\leq 0.05$) feature.

In *cross-validation*, a model is repetitively trained and tested with random splits of the data from a single release. We used stratified sampling when randomly splitting the data set to ensure equal proportion of vulnerable and neutral functions/files was maintained between the training and testing splits. In our study, we performed 10 repetitions of a 10-fold cross-validation. In other words, we trained and tested 100 models in each of the 23 releases of FFmpeg and Wireshark. The performance metrics—precision, recall, and $F^2$-measure—were aggregated across the 100 models.

[8] `https://scitools.com/understand/`

In *next release validation*, a model is trained with historical vulnerability data and tested by attempting to predict *known* future vulnerabilities. For example, consider a scenario where FFmpeg version 1.1.0 is being prepared for release. Retrospectively, all functions fixed for a post-release vulnerability in releases leading up to, and including, 1.1.0 are used in training the model. The model is then used to predict functions that are likely to require a fix for a post-release vulnerability in the future. The predictions are validated by comparing them against all functions known (in the context of this study) to be fixed for a post-release vulnerability in patch releases from the 1.1.x branch (i.e. FFmpeg releases 1.1.1 to 1.1.16).

While cross-validation is an acceptable, and commonly used [145, 160, 136, 38], approach, next release validation is intuitive and closer to reality. We chose to use both approaches to ensure that the performance of our models can be compared with those from prior vulnerability prediction literature.

Vulnerabilities are rare; the act of predicting vulnerabilities in software systems has been compared with searching for a needle in a haystack [160]. As an example, on average, a mere 0.67% of functions in FFmpeg were fixed for a vulnerability. At a file-level, however, an average of 3.47% of files in FFmpeg were fixed for a vulnerability. Predicting at a higher level of granularity may partially alleviate the problem of disproportionately sized populations of vulnerable and neutral entities. However, even at the file-level, the number of vulnerable entities are so few that the prediction models may be biased toward neutral entities resulting in a considerably high false negative rate. We have used a popular approach to dealing with class imbalanced data sets called SMOTE [18]. SMOTE uses synthetic over-sampling of the minority class (vulnerable functions/files) and a random under-sampling of the majority class (neutral functions/files). In our study, we have over-sampled vulnerable functions/files by 200% and under-sampled neutral functions/files by 200%.

## 4.5 Results

In the subsections that follow, we present the results from the empirical evaluation of our metrics.

### 4.5.1 RQ 1 - Association

Question: *Is a function/file more likely to be fixed for a post-release vulnerability if:*

*(a) it is near the attack surface or dangerous points?*

*(b) it has a higher probability of being traversed on a random walk from the attack surface?*

In this question, we wanted to understand if the proximity and risky walk metrics are capable of explaining the reality of historical post-release vulnerabilities. A statistically significant association between the metrics and historical post-release vulnerabilities support the utility of these metrics as early warning indicators of vulnerability likelihood.

We found a statistically significant association between the proximity metrics and vulnerable functions in 15 of the 16 FFmpeg releases and in all releases of Wireshark. The Cohen's $d$ effect size evaluation was predominantly *medium* in FFmpeg and *large* in Wireshark. The association results were consistent at the file-level as well, with the metrics being associated, to a statistically significant extent, with vulnerable files in 14 of the 16 FFmpeg releases and in all releases of Wireshark. The Cohen's $d$ effect size was predominantly *small* in both FFmpeg and Wireshark.

At both the function- and file-level, the median values of proximity metrics collected from vulnerable functions/files were lesser than that collected from neutral functions/files. In other words, vulnerable functions/files tend to be near the attack surface of a software system and also other functions/files regarded as dangerous.

| Vulnerable functions/files tend to be near the attack surface and/or dangerous points. |
|---|

The association analysis also revealed a statistically significant association between the risky walk metric and vulnerable functions in 13 of the 16 FFmpeg releases and in all releases of Wireshark. The Cohen's $d$ effect size evaluation was predominantly *small* in both FFmpeg and Wireshark. The association results were consistent with the file-level as well, with the metric being associated, to a statistically significant extent, with vulnerable files in 15 of the 16 FFmpeg releases and in all releases of Wireshark. The Cohen's $d$ effect size was predominantly *large* in both FFmpeg and Wireshark.

At both the function- and file-level, the median value of risky walk metric collected from vulnerable functions/files was higher than that collected from neutral functions/files indicating that vulnerable functions/files tend to have a higher probability of being traversed by a random walk starting at the attack surface.

| Vulnerable functions/files have a higher probability of being traversed on a random walk from the attack surface. |
|---|

### 4.5.2 RQ 2 - Prediction (Base)

Question: *Do proximity and risky walk metrics improve the performance of a base prediction model built with SLOC and coupling metrics?*

In this question, we wanted to understand if the proximity and risky walk metrics can be used in building a predictive model that can predict the likelihood of a function/file being vulnerable better than a base model built with SLOC and coupling metrics.

We begin the regression analysis by assessing the correlation between the different metrics: proximity, risky walk, SLOC, fan in, and fan out. We used the non-parametric Spearman's Rank Correlation Coefficient, $\rho$, to asses the correlation between metrics. We observed a very high positive correlation ($\rho \geq +0.97$) between the proximity metrics. The high correlation suggests that entry points, exit points, and dangerous points tend to be close to one another. As a consequence, only one of the three proximity metrics may be sufficient in explaining all three phenomena, rendering the other two metrics redundant. The correlation analysis also revealed a strong positive correlation ($\rho \approx 0.76$) between SLOC and fan out. The positive correlation between SLOC and fan out is understandable in that a function that has more SLOC is likely to have more function calls. We also observed a moderate negative correlation ($\rho \approx -0.65$) between fan out and the proximity metrics. All correlations were statistically significant with p-value $< 0.05$.

We chose to not remove the redundant features manually but to use regression analysis approaches that are capable of dealing with multicollinearity. We explored several parametric and non-parametric regression analysis approaches and found the random forest machine learning approach to perform the best. The model performance results presented here, and in RQ3, are from the random forest model. Furthermore, the performance metrics—precision, recall, and $F^2$-measure—presented here are the mean values of those obtained from the models using the cross-validation and next release validation approaches.

**FFmpeg**

*File-level:* The average values of precision, recall, and $F^2$-measure of the base model were 0.0467, 0.7938, and 0.1840, respectively. The random forest model outperformed the base model model with an average precision of 0.1138 (an increase of 143.47%) and average $F^2$-measure of 0.3196 (an increase of 73.69%). However, the average recall of the random forest model was 0.5753 (a decrease of 27.52% from that of base).
*Function-level:* The average values of precision, recall, and $F^2$-measure of the base model were 0.0114, 0.7200, and 0.0533, respectively. The random forest model outperformed the base model with an average precision of 0.0156 (an increase of 36.33%) and average $F^2$-measure of 0.0725 (an increase of 36.07%). However, the average recall of the random forest model was 0.5493 (a decrease of 23.71% from that of base).
**Wireshark**
*File-level:* The average values of precision, recall, and $F^2$-measure of the base model were 0.1147, 0.7574, and 0.3399, respectively. The random forest model outperformed the base model model with an average precision of 0.1841 (an increase of 60.53%) and average $F^2$-measure of 0.3646 (an increase of 7.25%). However, the average recall of the random forest model was 0.5250 (a decrease of 30.68% from that of base).
*Function-level:* The average values of precision, recall, and $F^2$-measure of the base model were 0.0245, 0.5259, and 0.1023, respectively. The random forest model outperformed the base model with an average precision of 0.0333 (an increase of 31.11%), average recall of 0.5991 (an increase of 13.93%), and average $F^2$-measure of 0.1294 (an increase of 26.55%).

| |
|---|
| The random forest model outperformed the base model in terms of $F^2$-measure at both function- and file-levels. |

As seen above, lowering the granularity of the metrics from file-level to function-level makes finding the "needle in a haystack" an order of magnitude more difficult. While function-level prediction has its challenges, the benefits warrant the need for more research at function-level prediction. For instance, in FFmpeg, if a file is predicted as vulnerable, developers must audit 306 SLOC, on average, however, if a function is predicted as vulnerable, developers must audit 30 SLOC, on average–a considerable reduction in effort.

### 4.5.3 RQ 3 - Prediction (Prior)

Question: *How do the prediction models built with proximity and risky walk metrics compare with prior vulnerability prediction literature?*

While traditional bug prediction is related to this question, our recent work has shown that the best bug prediction models would perform poorly when predicting vulnerabilities [104]. Further discussion on this is in Related Work in Section 4.7. Furthermore, vulnerability data used in building the model is one of the most important aspects in vulnerability prediction [77]. There have been many studies [89, 148, 37, 36] that use warnings from static code analyzers as indicators of vulnerability in a file. While these studies show that there is a correlation between warnings from static code analyzers and vulnerabilities, the correlation is moderate at best. In our prediction models, we have used real-world vulnerabilities i.e. those that were publicly disclosed, acknowledged, and fixed by the development team. In contrast, a recent work by Scandariato et al. [127] used files marked as vulnerable based on warnings from a static code analyzer. While the precision and recall of the model was shown to be high ($\geq 0.8$), the response from the model was not the likelihood of a file being vulnerable but its likelihood of having a static analysis warning.

Prediction models from prior vulnerability prediction literature have operated at file-

level [110, 82], component-level [111], or binary-level [160, 145]. We found only one vulnerability prediction model that attempted to predict vulnerabilities at a function-level [136]. As a consequence of the disparity in granularity, choice of study subjects, and vulnerability data used in building the model, a direct comparison of the performance of the models in terms of metrics like precision, recall, etc., may be unfair. However, since we have collected our metrics, and built the prediction models, at both the function- and file-levels, we have partially alleviated the limitation of direct comparison imposed by granularity. To ensure fairness, we compared our function- and file-level models with other function- and file-level models from prior literature, respectively. We also collected and compared the same metrics (e.g. precision, recall, etc.) that were used in the performance evaluation of the models from prior literature.

At the function-level, the random forest model, fitted to data from both FFmpeg and Wireshark, outperformed the logistic regression model proposed by Shin and Williams [136]. The model proposed by Shin and Williams [136] achieved an almost zero average false positive rate (FPR) but a considerably high average false negative rate (FNR). A low FPR and a high FNR suggests that the model may have marked almost all functions as neutral. The best of our random forest models achieved a considerably lower average FNR of 0.3610 (a 58.02% decrease from 0.86) while maintaining an acceptable level of average accuracy at 0.8995 (a 4.31% decrease from 0.94).

Our file-level random forest model, fitted to data from both FFmpeg and Wireshark, outperformed the file-level prediction model proposed by Theisen et al. [145]. The best of our random forest models had an average recall that was considerably higher at 0.5796 (a 1059.24% increase from 0.05), the precision was 0.1984 (a 71.24% decrease from 0.69). However, in vulnerability prediction we prefer higher recall over a higher precision [87]. Furthermore, the model proposed by Theisen et al. was built using only those files that ever appeared on stack traces from system crashes. There may be files with latent vulnerabilities (see [84]) that may have never crashed but their model does not consider these files. Putting such a model into operation means that the system must be in production, and potentially vulnerable, for a long time to get the stack trace data in the first place.

The file-level random forest models, fitted to data from both FFmpeg and Wireshark, outperformed the component-level prediction model proposed by Gegick et al. [38]. The CART model proposed by the authors achieved a recall of 0.57 but suffered a FPR of 0.48. The best of our random forest models had an average recall of 0.5796, which is similar to the model proposed by Gegick et al., however, the average FPR was 0.0876 (a 81.37% decrease).

In summary, both the function-level and file-level prediction models proposed in our work outperformed comparable models from existing vulnerability prediction literature. The true value of our metrics is in providing fine-grained, actionable, and interpretable intelligence about potential security risks that tend to vary with everyday changes to the source code. Furthermore, while performance metrics such as precision and recall provide a common ground to compare models, they fail to capture the nuances of the model building process.

| The random forest model, at both function- and file-levels, outperformed comparable models from prior literature. |
|---|

## 4.6 Limitations

Our empirical analysis is based on historical vulnerabilities, which are by no means comprehensive. Thus, many vulnerabilities may exist in our systems that have not been found.

This is a common limitation in empirical security research, and is the reason we use the word "neutral" instead of "not vulnerable".

A call graph is only an approximation of the system's function calls because ensuring all possible paths of control flow are represented is, at best, time consuming, and, at times, impossible. Program analysis researchers [42, 43] have proposed several call graph construction algorithms that produce call graphs with varying levels of precision. Researchers have used missing functions and/or function calls when comparing two call graphs [64] or two call graph generation tools [107]. In our study, we conducted manual inspection of the call graph to ensure its accuracy, and we suggest some added heuristics—number of fragments and monolithicity—to help users of our metrics understand how "close" they may be to getting as many graph edges and nodes as they will get.

The proposed metrics, especially the risky walk metric, depends on several parameters that, if not tuned properly, may result in poor risk analysis. We have conducted sensitivity analysis via parameter tuning across our models (See Appendix B). We found similar parameter values across our two case studies, indicating that our discovered parameters may generalize. Nonetheless, we recommend that some careful consideration be put into parameters when deploying these metrics.

Furthermore, the risky walk metric is sensitive to the weight assigned to edges in the call graph. While the sensitivity may seem to be a limitation of the metric, it indeed presents the users with an opportunity to configure the metric to better reflect the structure and history of a software system. For instance, in our empirical evaluation, we have chosen to increase the weight of edges terminating at historically vulnerable functions/files with the assumption that an attacker may attempt to start the reconnaissance by looking at such functions/files. We did, however, explore the impact of *not* weighting the edges terminating at historically vulnerable functions/files and found that the prediction models suffered a small (2.70% in FFmpeg and 7.84% in Wireshark) decrease in the average $F^2$-measure, while still outperforming the base model. The exploration revealed that the prediction performance of the models can be improved by assigning appropriate weights to the call graph edges thus allowing the users to customize the metric to their software systems.

## 4.7 Related Work

The attack surface as a metaphor for risk is far from new. Michael Howard of Microsoft proposed the idea of quantifying security of a software system by measuring its *attack profile* [45]. Michael Howard's proposal was to reduce the attack profile of a product by having only the most commonly used features enabled by default. He introduced the notion of *attackability* of a product as a measure of its exposure to an attack. He computed the Relative Attack Surface Quotient (RASQ) to compare the *attackability* of seven versions of the Windows operating system to assess the relative security between them. The notion of *attackability* was redefined by Howard et al. along three dimensions: targets and enablers, channels and protocols, and access rights [47]. Howard et al. also proposed a formal method of measuring attack surface of software system in terms of its attack vectors (features that may be used in an attack).

The granularity of attack surface measurement was lowered from a system-level to a design-level by defining entry points and exit points to identify resources that compose the attack surface [71]. The notion of *size* of the attack surface emerged as measured by number of entry and exit points. We propose the refinement of the attack surface metrics by extending them to individual functions (and files) and taking into account the structure

as they connect to the system's entry and exit points.

In addition to interpreting the attack surface as being the outer shell of a system, we could also consider all resources "exposed" through the surface as being part of the attack surface as well. Younis et al. [154, 155] used reachability analysis to assess the severity of a vulnerability and the probability that a vulnerability will be exploited. While this work used call graphs, they did not apply attack surface metrics to individual functions as we did. We also take dangerous system calls into account in our empirical analysis. Theisen et al. [145] used the attack surface metaphor to improve existing vulnerability prediction models. The authors have shown that approximating the attack surface of a software system using functions from stack traces improves the performance of existing vulnerability prediction models. The prediction models are at the source file and compiled binary levels. While their study focuses on prediction improvement, our study focuses on producing a more lightweight approach to collecting metrics that rely on the call graph and entry/exit points, and does not require a database of millions of stack traces from production usage data.

A vulnerability is a special kind of a software bug, one that has security consequences. Naturally, one may assume that bug/defect prediction models [150, 88, 161] may be used in vulnerability prediction. While an empirical connection has been observed [138], we have shown that bugs do not foreshadow vulnerabilities [104]. Thus, while the vulnerability prediction methods may resemble bug prediction, the models do not directly translate.

## 4.8 Summary

The goal of this study is to assess security risk through an empirical understanding of the relationship between vulnerabilities, individual functions/files, and the attack surface of a software system. We proposed novel attack surface metrics—proximity and risky walk—defined on the call graph representation of a software system. Our empirical analysis revealed a statistically significant association with historical vulnerabilities (RQ1), and that prediction models outperformed a base prediction model built with SLOC and coupling metrics (RQ2). Prediction models that leverage our metrics, at both function- and file-levels, outperformed comparable models from prior vulnerability literature (RQ3). We envision the metrics to be beneficial to both researchers and practitioners as they are simple to collect, intuitive to understand, and flexible to apply.

In the future, we will explore the space of personalization and edge weighting schemes for the call graph as that affords an enormous opportunity for configuration and room for innovation. Future studies can also examine how these metrics fare on non-monolithic systems, such as APIs.

# Chapter 5

# Systematization of Vulnerability Discovery Metrics

Software metrics help developers discover and fix mistakes. However, despite promising empirical evidence, vulnerability discovery metrics are seldom relied upon in practice. In prior research, the effectiveness of these metrics has typically been expressed solely with precision and recall of a prediction model. These prediction models, being black box machine learning models, may not be deemed useful by developers. However, by systematically interpreting the models and metrics, we can provide developers with nuanced insights about factors that have led to security mistakes in the past. In this study, we systematically review the literature to enumerate the vulnerability discovery metrics proposed and the extent to which their ability to support decision making has been validated. We identified 172 vulnerability discovery metrics from the literature but their decision-informing ability was primarily validated using association, discrimination, and prediction, with limited attention given to actionability, causality, and developer perception. We collected ten metrics (churn, collaboration centrality, complexity, contribution centrality, nesting, known offender, source lines of code, # inputs, # outputs, and # paths) from six open-source projects. We assessed the generalizability of the metrics across two contextual dimensions (application domain and programming language) and between projects within a domain, computed thresholds for the metrics using an unsupervised approach from literature, and assessed the ability of these unsupervised thresholds to classify risk from historical vulnerabilities in the Chromium project.

A part of the study presented in this Chapter is published in the Proceedings of the 2019 International Workshop on Data-Driven Decisions, Experimentation and Evolution (DDrEE) [102].

## 5.1 Motivation

Vulnerability discovery metrics [110, 109, 135, 136, 133, 137, 134, 138, 22, 81, 82, 85, 160, 127, 149, 115, 156], having been empirically-validated to be associated with historical vulnerabilities, have incredible potential to provide insights into the engineering failures that may have led to the introduction of vulnerabilities. Despite empirical evidence of the association between metrics and vulnerabilities, their adoption in practice has been limited owing to concerns such as the high frequency of false positive from predictive models that use the metrics as features, the granularity at which the metrics operate, and lack of interpretable and actionable intelligence from the metrics [91].

In prior vulnerability discovery metrics literature, there seems to be an overwhelming emphasis on optimizing precision and recall of vulnerability prediction models at the expense of interpretability, usability, and actionability of the insights from such models. However, by using the performance of a vulnerability prediction model to infer the utility of metrics, we are ignoring the metrics' ability to tell a story, as Fenton and Neil suggested in their software metrics roadmap almost twenty years ago [32]. Even if we had a vulnerability prediction model with a precision and recall of 90% in one project, we may not be able to apply it to predict vulnerabilities in a different project [162]. The insights from such a model and the metrics, however, may be transferable as being indicators of engineering failures that may have led to vulnerabilities in the past. For instance, consider a metric that identifies if a file is on the approximated attack surface [145] or one that quantifies the proximity of a function to the attack surface (Chapter 4). These metrics may not yield a marked improvement in precision and recall but provide valuable insights to contextualize the change that developers make to the software. In effect, we must ask ourselves *what is the metric telling us?* and *what can we ask developers to do?* We must *humanize* the metrics such that they can communicate the rationale for a source code entity to be considered vulnerable. In interpreting the feedback that a metric provides, developers gain awareness of the potential factors that lead to vulnerabilities thus aiding the inculcation of an attacker mindset.

Our research vision, as stated in Chapter 1, is *to assist developers in engineering secure software by providing a technique that generates scientific, interpretable, and actionable feedback on security as the software evolves* [97]. In this chapter, we describe an essential step toward the accomplishment of this research vision through (1) the systematization of the state of the art in vulnerability discovery knowledge, specifically, in the realm of metrics-based discovery of vulnerabilities and (2) the assessment of the potential for deriving metrics- and data-driven insights from vulnerability discovery metrics in an unsupervised way.

We address the following research questions:

**RQ 1 - Enumeration**

What metrics have been proposed to discover security vulnerabilities in software?

**RQ 2 - Validation**

How have researchers evaluated the decision-informing ability of the metrics to discover security vulnerabilities in software?

**RQ 3 - Generalizability**

Are vulnerability discovery metrics similarly distributed across projects?

**RQ 4 - Thresholds**

Are thresholds of vulnerability discovery metrics effective at classifying risk from vulnerabilities?

## 5.2 Methodology

In this section, we describe the methodology used in collecting and analyzing the data needed to address the research questions. Addressing RQ 1 and RQ 2 requires a traditional systematic literature review methodology, while addressing RQ 3 and RQ 4 requires a traditional empirical research methodology. We describe the methodology here in a way that is as decoupled from the research questions as possible. We will, however, refer to specific parts of this section when presenting the results for each research question.

### 5.2.1 Systematic Review

In this section, we describe the methodology used to collect and analyze the data to address research questions RQ 1 and RQ 2.

> *Sidenote: At the outset, we want to acknowledge an event that caused us to deviate from the systematic literature review methodology presented in this section. While we were being methodical in developing the protocol to guide the systematic literature review, our colleague—Patrick Morrison—from North Carolina State University published a study mapping the field of security metrics [92]. As a result, the initial few steps of our methodology were rendered redundant. We adapted our methodology to use the studies mapped by Morrison et al. as our candidate studies thus removing the need to perform the search for candidate studies ourselves.*

A typical [55] systematic literature review has three main stages: (1) plan, (2) conduct, and (3) report. In each of these main stages, there are several subtasks that must be accomplished to ensure that the review is carried out in as rigorous, unbiased, and repeatable way as possible. We describe the subtasks in the planning stage of the systematic review in the remainder of this section. The planning stage concludes with the publication of a review protocol to the research team. The conducting stage involves carrying out the plan outlined in the review protocol and the reporting stage involves presenting the observations from the review. We describe the conducting and reporting stages when presenting our results in in Section 5.3.

In the planning stage, the specifics of all aspects needed for the conduct of the systematic review being proposed are identified and documented. The planning stage is crucial to ensure the rigor of the systematic review. In the subsections that follow, the various subtasks of the planning stage are described in detail. The stage begins by determining if there is indeed a need for the review being proposed. The stage concludes with the publication of a review protocol to the research team. The research team will then use the protocol to guide the conduct of the systematic review, revisiting the planning stage only when there are issues identified with the protocol during the conduct stage.

The protocol has been developed in adherence with the guidelines for performing Systematic Literature Reviews in Software Engineering prescribed by Kitchenham and Charters [55]. In addition to the guidelines prescribed by Kitchenham and Charters, specific aspects

of the the review protocol have further been informed by other, more specific, guidelines. For instance, we used the guide published by Counsell [25] to aid formulating the research questions in our systematic review. We reference such specific guidelines in the sections that they are applicable in.

**Need for the Review**

The literature on vulnerability discovery metrics is largely scattered across many primary studies [110, 109, 135, 136, 133, 137, 134, 138, 22, 81, 82, 85, 160, 127, 149, 115, 156]. To aid developers in engineering secure software, we must characterize the factors that may have led to the introduction of vulnerabilities in the past. The primary need for the systematic review is to aggregate the empirical evidence demonstrating the utility of metrics to discover vulnerabilities in software. The secondary need for the systematic review is as a prelude to the subsequent research studies conducted toward achieving our research vision.

As recommended by the guidelines [55], we searched for existing secondary studies addressing the same (or similar) research questions as we aim to address in our systematic review. In this search, we used Google Scholar[1] as the query engine and `software + vulnerability + review` as the search string. While we did not find any secondary studies addressing the same research questions as we aim to address, we did find two studies [67, 39] that seemed related. However, these studies are not systematic review but rather surveys of the different approaches to discover vulnerabilities. The use of metrics to discover vulnerabilities in software was just one of many approaches described in the work by [39].

**Research Question(s)**

The guidelines [55] prescribe the use of the PICOC (Population, Intervention, Comparison, Outcomes, and Context) criteria to structure research questions in systematic reviews. The components (i.e. PICOC) of the research question are determined by the type of research question. The type of research question is in-turn determined by the type of evidence available in the primary studies (See Table 3 in the study by Fineout-Overholt and Johnston [33] for examples of different types of questions and the types of evidence needed to answer a given question).

In the context of our systematic review, these components of the research question are as follows.

**Population** The population component in our review is *software* since the goal is to understand the metrics that aid in discovery of vulnerabilities in software.

**Intervention** The intervention component in our review is the *increasing or decreasing of the value of an empirically-validated metric* as a result of some activity in the Software Development Lifecycle.

**Comparison** The comparison component is *not applicable* in our review because the goal of our review is not to compare the intervention across studies but to characterize the intervention itself.

**Outcomes** The outcome that concerns our review is *discovery of security vulnerabilities.*

**Context** In Software Engineering, the context is the context in which the primary study was conducted. The context could include factors such as where (academia or industry), what (open source or closed source), who (if applicable, practitioners, academics, or students), and how (quantitative or qualitative). In our review, the only restriction on context is that the primary study must be quantitative. By not restricting the where,

[1] `https://scholar.google.com/`

what, and, who, we are likely to get a holistic perspective on the literature increasing the potential for our review to produce more generalizable conclusions.

The research questions RQ 1 and RQ 2 were formulated using the aforementioned guidelines.

**Search Strategy**

The approach to identify the primary studies that will be systematically reviewed to address the research questions is outlined here. The search for primary studies is accomplished by search strings to search various source of candidate studies. There are two steps that must be accomplished before the search can begin, they are: (1) identify sources of primary studies and (2) identify search keywords.

A note on terminology before we proceed: we use the phrase *candidate study* to refer to any study that is *hypothesized* to be relevant to our review according to some predetermined criteria (i.e. is retrieved by the keywords used to search). The candidate studies are subject to thorough inclusion and exclusion criteria to identify those studies that are relevant to address the research questions posed in our systematic review. We use the phrase *primary study* to refer to a candidate study that satisfies the inclusion and exclusion criteria.

We use a set of primary studies known, from manual search and prior experience, to be relevant to address the research questions posed in our systematic review to identify sources of primary studies and validate the search keywords. The set of known primary studies is known as the Quasi-Gold Standard Set as described by Zhang *et al.* [159]. The primary studies that compose the Quasi-Gold Standard Set in our review are listed in Appendix C.

We used the Quasi-Gold Standard approach prescribed by Zhang *et al.* [159] to validate, and improve, the effectiveness of the final search string. More specifically, we use the *Quasi-Sensitivity* metric defined in Equation (5.1). The metric enables objective assessment of the effectiveness of the search strings. For instance, assume that the Quasi-Gold Standard Set contains 10 studies published by ACM. If a search using the ACM Digital Library retrieved 8 (of the 10) studies, then the Quasi-Sensitivity is 80%.

$$\text{Quasi-Sensitivity} = \frac{\#\text{ Studies in Quasi-Gold Standard Set Retrieved}}{\text{Total }\#\text{ Studies in Quasi-Gold Standard}}\% \tag{5.1}$$

As Zhang *et al.* [159] have suggested, a rational threshold for Quasi-Sensitivity may be assumed to say that the search strategy is acceptable. We have chosen 80% as the threshold in our review.

*Identify Sources of Primary Studies*

We begin by identifying the sources from which our primary studies are likely to originate. Since we have a set of known primary studies (the Quasi-Gold Standard Set), we simply enumerate the publishers of these primary studies. The publishers become the sources of primary studies in our review.

The studies in our Quasi-Gold Standard Set were published by five distinct publishers: (1) ACM, (2) IEEE, (3) Elsevier, (4) Springer, and (5) USENIX. Each of these publishers provide a service to search their respective publication databases. The name, and location (URL), of the search services used in our review are shown in Table 5.1.

In addition to the search services shown in Table 5.1, we used the private search service provided by Rochester Institute of Technology (RIT), called Summon, that is used by RIT Library (`https://library.rit.edu/`). We chose to include Summon as a catch-all source of primary studies to include studies that may be missed by the other search services. In

Table 5.1: Search services provided by publishers of academic content

| Publisher | Service (URL) |
|---|---|
| ACM | ACM Digital Library (`https://dl.acm.org/`) |
| IEEE | IEEE Xplore (`https://ieeexplore.ieee.org/`) |
| Elsevier | ScienceDirect (`https://www.sciencedirect.com/`) |
| Springer | SpringerLink (`https://link.springer.com/`) |
| USENIX | USENIX (`https://www.usenix.org/publications/proceedings`) |

the past, systematic reviews have used Google Scholar (`https://scholar.google.com/`) for similar purposes.

*Identify Search Keywords*

We followed the approach outlined below to identify the keywords that will be used to query sources of academic publications for candidate studies.

1. We started with the population (software), intervention (metrics), and outcome (discover vulnerabilities) components from the research questions outlined earlier. In addition to these components, we also included the context (quantitative) in which primary studies should have been conducted to be considered relevant.
2. We identified synonyms of the keywords pertaining to population, intervention, outcome, and context. We also considered truncated versions (stems) of the keywords to ensure variations in usage of the same keyword was accounted for. For instance, the truncated keyword "vulner" can capture both "vulnerability" and "vulnerabilities". When applicable, we also considered alternative spellings[2] of the keywords to allow for difference in word usage across English dialects.
3. We combined the search keywords, their corresponding synonyms, truncated forms, and alternatives spelling using the boolean `OR` operator.
4. We then used the boolean `AND` operator to combine the search strings (combination of search keywords, their synonyms, truncated forms, and alternative spellings) corresponding to population, intervention, and outcome.

The search strings obtained by applying the aforementioned approach are as follows.

**Population**

- "software" `OR` "application" `OR` "system" `OR` "program" `OR` "product" `OR` "code"

**Intervention**

- "metric" `OR` ("measure" `OR` "measurement")

[2] `https://en.wikipedia.org/wiki/Wikipedia:List_of_spelling_variants`

**Outcome**

- (("discover" OR "detect" OR "predict" OR "uncover" OR "locate") AND ("vulnerability" OR "vulnerabilities" OR "security vulnerability" OR "security vulnerabilities"))
- (("vulnerability" OR "security vulnerability") AND ("discovery" OR "detection" OR "prediction"))

**Context**

- "quantitative" OR "empirical" OR "evidence-based" OR "experiential"

The individual search strings corresponding to the population, intervention, outcome, and context are logically combined using the form (outcome AND intervention AND population AND context). The effective search string is shown in Figure 5.1

(((("discover" OR "detect" OR "predict" OR "uncover" OR "locate") AND ("vulnerability" OR "vulnerabilities" OR "security vulnerability" OR "security vulnerabilities")) OR (("vulnerability" OR "security vulnerability") AND ("discovery" OR "detection" OR "prediction"))) AND ("metric" OR ("measure" OR "measurement")) AND ("software" OR "application" OR "system" OR "program" OR "product" OR "code") AND ("quantitative" OR "empirical" OR "evidence-based" OR "experiential")

Figure 5.1: Effective search string in the systematic review

**Inclusion and Exclusion Criteria to Select Primary Studies**

By applying the search strategy, we will identify candidate studies that are hypothesized to be relevant to our systematic review. However, not all candidate studies may contain the data needed to address all the research questions posed in our systematic review. In this step of the review process, the candidate studies are subject to additional criteria (referred to as inclusion and exclusion criteria) to identify those studies that are relevant to our review. The inclusion and exclusion criteria used in our systematic review are described here.

Applying the inclusion and exclusion criteria to evaluate the relevancy of candidate studies is, for the most past, an objective exercise. As a result, the benefit of having at least two authors independently apply the inclusion and exclusion criteria to all the candidate studies may be trivial. However, to quantify the objectivity of applying the inclusion and exclusion criteria, a random subset of the candidate studies will be independently evaluated for relevancy by at least two authors. The level of agreement between the two authors will be quantified using the Cohen's $\kappa$. If the level of agreement is not almost perfect (See scale in a paper by Landis and Koch [60]), all candidate studies will be independently evaluated for relevancy by at least two authors to mitigate subjectivity. In the event that two or more authors apply the inclusion and exclusion criteria to all candidate studies, the level of agreement between the authors will be assessed using Cohen's $\kappa$, if two authors were involved, or Fleiss' $\kappa$, if more than two authors were involved. Any disagreements in applying the inclusion and exclusion criteria will be resolved through discussion among the authors involved. If the discussion yields no consensus, an additional author may be involved to mediate the disagreement.

*Exclusion Criteria*

The exclusion criteria defines rules to exclude a candidate study from consideration. The exclusion criteria in our review is as follows.

- Studies for which the published full text is inaccessible through the University's subscription.
- Studies not written in English.
- Studies that have not been subject to peer review.
- Studies published as extended abstracts or supplement to poster and/or presentation.
- Studies that are secondary or tertiary.
- Studies published before the year 2000.[3]
- Studies not published in a venue related to the discipline of Computer Science.

*Inclusion Criteria*

- Study proposes and/or evaluates one or more metrics as a means to discover vulnerabilities in software.
- Study presents empirical evidence when reasoning about the utility of the metrics to discover vulnerabilities in software.

  The empirical evidence presented is in the context of publicly-disclosed historical vulnerabilities either disclosed via the National Vulnerability Database (NVD) or as advisories or bulletins in vendor-specific security disclosure portals such as Microsoft Security Bulletin[4] and Mozilla Foundation Security Advisories.[5] The rationale behind this qualification is that we do not want to include studies that *assume* static analysis warnings to be indicative of real vulnerabilities and propose and/or evaluate metrics to discover static analysis warnings.
- Study describes the approach to collect the metric being proposed and/or evaluated with enough detail to enable replication.
- Study provides an interpretation of the metric as being a factor in vulnerability discovery.

**Data Extraction Strategy**

The data needed to address the research questions in our systematic review will be extracted from the primary studies using a predetermined data extraction form. The design of the data extraction form is driven by the research questions and the type of data needed to address the research questions. The data extraction form that will be used in our systematic review is presented in Table D.1 in Appendix D.

[3] The year threshold is based on a similar study by Morrison *et al.* [92]. The threshold, while arbitrary, is reasonable since the term *vulnerability* was formally defined by Krsul in his PhD thesis [57] in 1998. [4] `https://technet.microsoft.com/en-us/security/bulletins.aspx`
[5] `https://www.mozilla.org/en-US/security/advisories/`

The process of extracting data from primary studies, for the most part, is an objective exercise. As a result, the benefit of having at least two authors independently extract the same data from all primary studies may be trivial. However, to quantify the objectivity of the data extraction process, at least two authors will independently extract data from a random subset of the primary studies. The level of agreement between the two authors will be quantified using the Cohen's $\kappa$. If the level of agreement is not almost perfect (See scale in a paper by Landis and Koch [60]), at least two authors will independently extract data from all primary studies to mitigate subjectivity. In the event that two or more authors extract the data from all primary studies, the level of agreement between the authors will be assessed using Cohen's $\kappa$, if two authors were involved, or Fleiss' $\kappa$, if more than two authors were involved. Any disagreements in data extraction will be resolved through discussion among the authors involved. If the discussion yields no consensus, an additional author may be involved to mediate the disagreement.

**Extracted Data Synthesis Approach**

There are two ways of synthesizing data for addressing research questions in a systematic review: descriptive (narrative) synthesis and quantitative synthesis. Although primary studies in our systematic review are likely to be empirical in nature, using formal meta-analysis to synthesize data in a quantitative way may be infeasible since the protocol for reporting quantitative results tend to vary greatly between primary studies [16]. As a result, we will use the descriptive (narrative) synthesis approach to synthesize the data needed for addressing the research questions in our systematic review.

**Protocol Pilot**

The review protocol is perhaps the most important factor in ensuring that the systematic review is carried out in as rigorous, unbiased, and repeatable way as possible. The review protocol described thus far has been collaboratively developed by two authors to uncover any methodological flaws as early as possible. We also piloted the protocol by applying it to a subset of candidate studies identified using the search strategy.

### 5.2.2 Empirical Research

In this section, we describe the methodology used to collect and analyze the data to address research questions RQ 3 and RQ 4.

**Data Collection**

We collected nine of the 18 vulnerability discovery metrics described in Section A.1 of Appendix A from six open-source projects spanning three domains. Being a manually collected metric, offender was difficult and time consuming to collect from all six projects. Therefore, we only collected it for the Chromium project, reusing a considerable portion of the data collected as part of a previous work [106]. The projects that we considered in our study are shown in Table 5.2.

We only collected the metrics from file paths ending with the extensions `.c`, `.cc`, `.cpp`, `.cxx`, `.h`, `.hh`, `.hpp`, `.hxx`, or `.inl` for C/C++ projects and `.java` for Java projects.

**Data Analysis**

In the subsections that follow, we describe the series of analysis we performed on the vulnerability discovery metrics collected from the six open-source projects. We used `R` [119] version 3.5.1 to conduct our data analysis.

*Assessing Normality*

In using statistical methods, we must validate the assumption of normality to inform the choice of parametric or nonparametric statistical methods. We used the Anderson-Darling

Table 5.2: Projects from which the vulnerability discovery metrics were collected

| Domain | Project | Language | Size* |
|---|---|---|---|
| Browser | Chromium | C/C++ | 9,054,450 |
| | Firefox | C/C++ | 6,977,203 |
| Operating System | Linux | C/C++ | 13,101,179 |
| | OpenBSD | C/C++ | 9,147,222 |
| Application Server | Tomcat | Java | 326,748 |
| | WildFly | Java | 524,240 |

*Total number of source lines of code across all languages.

test to assess if the metrics collected from a project are normally distributed. We found, with statistical significance (p-value $\ll 0.001$), that no discrete- or continuous-valued metric was normally distributed in any of the projects.

*Assessing Generalizability*

In addressing RQ 3, we wanted to assess if the vulnerability discovery metrics proposed in the literature are generalizable. The interpretation of the term generalizability may be subjective. Therefore, we define the necessary (not sufficient) condition for generalizability as the need to have consistent distribution. We assessed the generalizability condition of a metric by comparing the distribution of the metric values collected from different projects. We used the same approach that Zhang *et al.* [158] used to assess the impact of contextual dimensions on the distribution of metrics.

We used the nonparametric Kruskal–Wallis to assess if the sample distribution of metric values collected from projects, grouped by contextual dimensions, originated from the same underlying distribution. We consider the outcome from the Kruskal–Wallis test to be statistically significant if p-value $< 2.78e-03 = 0.05/9/2$ ($\alpha = 0.05$ corrected for multiplicity using Bonferroni Correction).

A statistically significant outcome from Kruskal–Wallis test would indicate that at least one metric distribution is different from the rest. To explain the difference(s) further, we supplement the outcome from Kruskal–Wallis test with observations from pairwise comparison of distribution of metric values using the nonparametric Mann–Whitney–Wilcoxon test.

We consider the outcome from the Mann–Whitney–Wilcoxon test to be statistically significant if p-value $< 7.94e{-}04 = 0.05/63$ ($\alpha = 0.05$ corrected for multiplicity using Bonferroni Correction). The inference from a statistically significant outcome from Mann–Whitney–Wilcoxon test is that the metric distributions being assessed are different. To quantify the magnitude of the difference in distributions, we used Cliff's $\delta$ [69], a nonparametric effect size measure. The difference between distributions is considered negligible when $\delta < 0.147$, small when $0.147 \leq \delta < 0.33$, medium when $0.33 \leq \delta < 0.474$, and large when $\delta > 0.474$ [124].

In summary, we consider a metric to satisfy the generalizability condition if it is similarly distributed irrespective of contextual dimensions and between projects within a domain with negligible to medium effect size (same as the threshold of large effect size used by Zhang *et al.* [158]).

*Computing Thresholds*

Vulnerability discovery metrics are typically evaluated using a supervised approach requiring data on historical vulnerabilities to train a prediction model that uses the metrics as explanatory variables. While some projects may have a curated list of historical vulnerabilities to satisfy this prerequisite, a model trained with data from one project may not be directly applicable to discover vulnerabilities in a different project [162]. An unsupervised approach is, therefore, needed.

An intuitive interpretation of a metric is to establish, and use, thresholds to label an entity being measured to exhibit certain attribute to a higher or lower degree. While computing a universal threshold for all projects is intractable, Lanza and Marinescu suggest using statistical information rankings to determine explicable thresholds [61].

In our study, we used the methodology proposed by Alves *et al.* [5] to compute the thresholds. Despite being an unsupervised approach, the methodology proposed by Alves *et al.* was found to be effective in the prediction of fault-proneness in comparison with other supervised approaches [14].

We compute the thresholds for only those metrics that satisfied our generalizability condition. As proposed by Alves *et al.* [5], we compute the thresholds by taking metric values from all projects together and use the same 70%, 80%, and 90% of the weighted (using the source lines of code metric) metric value averaged over all projects to determine the risk levels. We use the following risk levels to assess risk using a metric ($m$): low ($m < 70\%$), medium ($70\% \leq m < 80\%$), high ($80\% \leq m < 90\%$), and critical ($m \geq 90\%$).

In addressing RQ 4, we use the metrics' thresholds to classify the risk from metrics collected from known (historically) vulnerable files (determined by the offender metric). We quantify the effectiveness of such risk classification by expressing the coverage (i.e. percentage of vulnerable files covered) of and assessing the odds of finding a vulnerable file in each of the non-trivial risk levels (i.e. risk levels other than low).

## 5.3 Results

In the subsections that follow, we address the research questions. We begin by providing an overview of the primary studies reviewed. As highlighted in the side note at the beginning of Section 5.2.1, we used the 70 studies already mapped by Morrison et al. as containing security metrics [92] as the candidate studies in our work. 5 of these 70 studies were already in our Quasi-Gold Standard Set. We applied our inclusion and exclusion criteria to the remaining 65 studies and identified 8 as being relevant. The 8 primary studies identified as being relevant through the review are listed in Appendix E. In total, we reviewed 26 primary studies (8 from review and 18 from Quasi-Gold Standard Set) spanning the ten years from 2007 to 2017. The data extracted from these 26 primary studies are available for download as a data set from Zenodo [103].

The 26 primary studies were authored by 15 distinct first authors. While all primary studies evaluated the utility of metrics to discover vulnerabilities, there were certain differences between the studies. These differences were in (1) the granularity at which the analysis was performed, (2) the metric used to quantify vulnerability of an entity (which in turn depends on the granularity of analysis), and (3) the subjects of study, and their respective nature (open-source or closed-source), used in the empirical analysis. An overview of the differences we observed among the primary studies follows.

**Level of Granularity**

We identified 9 distinct levels of granularity at which researchers have analyzed the utility of vulnerability discovery metrics. The level of granularity is an important aspect since higher level of granularity has been cited as one of the factors inhibiting the adoption of metrics in practice [91]. We observed *file*, *component*, and *commit* to be the top three most common level of granularity with 11, 5, and 4 primary studies analyzing the metrics at the file, component, and commit level, respectively. Insights at the commit level of granularity is likely to be the most actionable because (1) the insights are in the context of a logical unit of change to the software system and (2) the developer contributing the commit, being familiar with the change, is most suited to interpret the insights to assess its validity.

**Vulnerability Metric**

We identified 5 distinct metrics used to quantify vulnerability of an entity (which dependents on the granularity). The metric used to quantify vulnerability becomes the dependent (or response) variable when evaluating the utility of vulnerability discovery metrics. We observed that the boolean metric, *vulnerability*, was the most common measure of vulnerability of an entity with 16 primary studies using it as the dependent variable. The *vulnerability* metric is *true* if the entity has had (or has been fixed for) a vulnerability in the past, *false* otherwise.

**Subjects of Study**

We identified 87 distinct subjects of study in which the utility of vulnerability discovery metrics has been evaluated in. When a primary study (the study by Gegick *et al.* [36], for instance) did not report the name of the subject of study for confidentiality, we used the *unreported-closed* or *unreported-open* in place of the name of the subject of study for closed-source and open-source, respectively. Firefox, Apache httpd, and Wireshark were the most common subjects of study with 6, 5, and 4 primary studies that used these in the empirical analysis, respectively. We also observed that 86.21% of the subjects of study were written in C or C++, 10.34% in PHP, and 3.45% for which the language was not reported either because the subject of study was proprietary or because the specifics of the subjects were only presented in aggregate. As for the nature of the subjects of study, we observed that 22 primary studies considered only open-source projects in their analysis while the remaining 4 primary studies considered only closed-source projects. The bias toward open-source is expected given the inherent challenges in gaining access to analyzing closed-source projects. In fact, the 4 primary studies that only analyzed closed-source projects were authored by 2 distinct first authors and all 4 primary studies had at least one collaborator from the software development organization whose projects were analyzed (Microsoft and Cisco in case of the 4 primary studies).

As we described in Chapter 1, one of the key contributions of our work is the implementation of the vulnerability discovery metrics identified through the systematic review. While our initial goal was to implement all the metrics identified, we found ourselves having to trade off between making contribution to knowledge versus community. As a consequence of weighing this trade off, we chose to implement a subset of the metrics identified during the systematic review. However, our decision to implement the metrics as containerized microservices led to the creation of a platform which provides all necessary architectural boilerplate to implement additional metric services with minimal effort. The platform is called Samaritan and the specifics of the platform and the subset of vulnerability discovery metrics implemented to date are presented in Appendix A.

### 5.3.1 RQ 1 - Enumeration

Question: *What metrics have been proposed to discover security vulnerabilities in software?*

**Motivation.** The purpose of the systematic literature review is to systematize the vast vulnerability discovery metric knowledge [92] scattered across the many academic publications [110, 109, 135, 136, 133, 137, 134, 138, 22, 81, 82, 85, 160, 127, 149, 115, 156]. The motivation for the enumeration research question is to enumerate all metrics that have been shown, through empirical evaluation, to be utilitarian in vulnerability discovery.

**Approach.** We used the data extraction form presented in Table D.1 in Appendix D to systematically collect data needed to address this research question from each primary study. While collecting data about the metrics proposed and/or evaluated in each primary study, we observed that the same metric was aggregated using different aggregation functions in different primary studies. For instance, while Chowdhury and Zulkernine [21] aggregated Cyclomatic Complexity metric at the file level by computing the *average* of the metric value collected from all functions in the file, Shin and Williams [137] aggregated the same metric at the file level by computing the *sum* and *maximum* of the metric value collected from all functions in the file. Since we used Airtable, a spreadsheet-database hybrid, to implement the data extraction form, we were able to normalize the extraction form such that we collected unique metrics separately and used references to associate metrics to a primary study. Furthermore, unlike Morrison *et al.* [92], who clustered metrics related to one another into distinct groups to deduplicate security metrics in the literature, we treat two (or more) metrics to be the same if and only if their implementation is the same or if one metric can be obtained by aggregating another metric. For instance, line churn and thirty-day line churn that appear in the study by Meneely *et al.* [85] are treated the same because the implementation of the thirty-day line churn metric is simply a filtered aggregation of the line churn metric. However, we have treated percentage of interactive churn and thirty-day percentage of interactive churn that also appear in the study by Meneely *et al.* [85] as two separate metrics because their implementations are different, albeit slightly, and one metric cannot be aggregated from the other.

**Observations.** We identified a total of 172 metrics that have been proposed and/or evaluated for as factor in discovering vulnerabilities. The list of metrics identified is presented in Appendix F. In terms of the data type of the metric, almost all (159) metrics were numerical with a small number (11) being boolean. We had a one metric each of type categorical (programming language from the study by Perl *et al.* [115]) and tuple (attack surface measurement from the study by Manadhata and Wing [71]).

On average, 11 metrics were proposed and/or evaluated in each primary study with the study by Zimmermann *et al.* [160] having the most metrics at 26. We observed that the source lines of code was the one metric that was in a majority of the primary studies (14 of the 26). The maximum nesting level of control structures in a function and the incoming information flow (aka fan in) were a close second, appearing in 8 primary studies each. However, only 28 of all identified metrics have been studied by two or more distinct first authors. As mentioned earlier, more than 95% of the subjects of study were written in C, C++, or PHP. We observed that only 6 metrics have been studied in the context of all three programming languages. We also observed that 14 metrics have been studied in both open-source and closed-source studies.

172 metrics have been proposed and/or evaluated to discover vulnerabilities in software with 28 of these metrics studied by two or more distinct first authors. Over 95% of the subjects of study used in the literature were written in C, C++, or PHP and only 6 metrics

identified have been studies in the context of all three programming languages.

The observations from the enumeration research question highlights a key limitation that must be overcome if we were to attempt to use the vulnerability discovery metrics to assist developers in engineering secure software: the considerable majority of the metrics, having been validated in the context of a single subject of study, do not seem to have the potential for deriving generalizable insights to support vulnerability discovery. We hope that the SAMARITAN metric services (described in Appendix A) alleviates the burden of implementing algorithms to collect metrics from subjects and encourages researchers to replicate the metrics in the context of diverse subjects.

### 5.3.2 RQ 2 - Validation

Question: *How have researchers evaluated the decision-informing ability of the metrics to discover security vulnerabilities in software?*

**Motivation.** In the enumeration research question, we merely enumerated the vulnerability discovery metrics identified by our review. The motivation for the validation research question is to characterize the extent to which the decision-making ability of these metrics has been validated in the literature.

**Approach.** We used the data extraction form presented in Table D.1 in Appendix D to systematically collect data needed to address this research question from each primary study. The data extraction form was structured as a matrix with rows being the primary studies and columns being the validation criterion. We used the value "Yes" in a cell corresponding to row (`r`) and column (`c`) if the primary study in `r` subjected the metrics being studied to the validation criterion in column `c`. Since the motivation for this research question is to characterize the ability of vulnerability discovery metrics to inform developer decisions, we used the set of ten atomic validation criteria[6] reported to have *decision-informing* advantage [84]. The ten atomic validation criteria considered are: actionability, association, causal model validity, causal relationship validity, constructiveness, discriminative power, economic productivity, prediction system validity, rank consistency, and trackability.

We used the definition of each of the ten validation criterion as reported by Meneely et al. [84] in assessing if a primary study used a specific criterion. While identifying evidence to support the use of some of the criteria is straightforward, there is an element of subjectivity involved in the rest. We overcame this limitation by developing an interpretation guide for each of the ten validation criterion before we started to collect the data from the primary studies. For instance, the interpretation guide for the actionability criterion involved searching for the mention of interpretive guidelines and/or suggested actions associated with the metrics in a primary study.

In addition to the validation criteria, we also collected one more piece of data from each of the primary studies: a boolean indicating if the researchers contacted real developers and/or experts to validate the metrics.

**Observations.** The percentage of primary studies that subjected vulnerability discovery metrics to each of the ten atomic validation criteria reported to have decision-informing advantage [84] is shown in Table 5.3. As can be inferred from the table, there seems to be a bias toward validating metrics using prediction system, association, and discriminative power. The observation, while unsurprising, highlights an key aspect that is seldom considered when validating metrics: the ability of the metrics to tell a story, as Fenton and Neil

[6] An atomic validation criterion is one that cannot be subdivided further.

Table 5.3: Percentage of primary studies that subjected vulnerability discovery metrics to each of the ten atomic validation criteria reported to have decision-informing advantage [84]

| Criterion | % Primary Studies |
|---|---|
| Actionability | 23.08% |
| Association | 57.69% |
| Causal Model | 0.00% |
| Causal Relationship | 0.00% |
| Constructiveness | 73.08% |
| Discriminative Power | 57.69% |
| Economic Productivity | 23.08% |
| Prediction System | 80.77% |
| Rank Consistency | 26.92% |
| Trackability | 3.85% |

[32] suggested in their software metrics roadmap almost twenty years ago. Fenton and Neil suggested the use of causal modeling to describe the relationship between a metric and a quality attribute (vulnerability, in our case).

We also observed that in only one (the study by Perl *et al.* [115]) of the primary studies we reviewed, the authors reported to have reached out to real developers to validate the predictions from their model. Again, the observation is unsurprising given the resource intensiveness of validating metrics/models with real developers.

> The validation of vulnerability discovery metrics in the literature is biased toward using prediction system, association, and discriminative power with limited involvement of real developers.

The observations from the validation research question highlights the need to explore validation criteria beyond the conventional association, discrimination, and prediction. We hope that the Samaritan metric services (described in Appendix A) alleviates the burden of implementing algorithms to collect metrics from subjects and encourages researchers to explore a more holistic approach to validating the metrics.

### 5.3.3 RQ 3 - Generalizability

Question: *Are vulnerability discovery metrics similarly distributed across projects?*

**Motivation.** As we observed when addressing the 1 research question, the literature has no shortage of vulnerability discovery metrics with over 172 identified through our systematic literature review. However, about 83.72% of these metrics have been empirical evaluated solely by the authors who proposed them. Furthermore, the isolation of the empirical eval-

uation to a few subjects of study with little diversity in programming language limits the potential for generalizability of the metrics. A key first step toward realizing the (near) universal adoption of these metrics is to assess if these metrics are generalizable across multiple projects. The assertion being that the insights that one can gain from a generalizable metric can be transferred from one project (perhaps one that has had engineering failures in the past) to another.

The motivation for the generalizability research question is to assess if security metrics, particularly vulnerability discovery metrics, are generalizable across projects.

**Approach.** The interpretation of the term generalizability may be subjective. In our study, we use the similarity in distribution of metrics across projects as a proxy for generalizability. The approach to address the research question involved supplementing insights from Krukal–Wallis test with that from Mann–Whitney–Wilcoxon test and Cliff's $\delta$.

**Observations.** We use a traditional violin plot to compare the distribution of the nine discrete- and continuous-valued metrics in the six projects considered in our study. The plot provides us an inkling of the metrics that are likely to be generalizable. Shown in Figure 5.2 is a plot comparing the distribution of churn and collaboration metrics.[7] As can be inferred from the figure, churn appears to be similarly distributed across all projects (irrespective of domain and language). However, collaboration centrality seems to be similarly distributed between projects in the browser domain but differently distributed between projects in both the operating system and application server domains and, consequently, between domains.

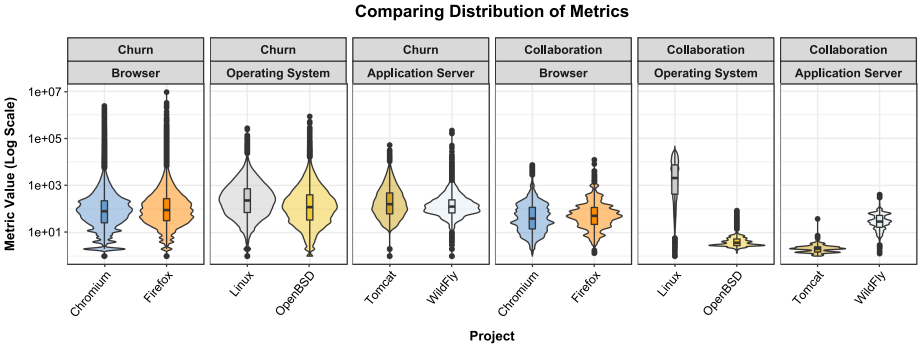



Figure 5.2: Comparing the distribution of churn and collaboration metrics across all projects considered in our study

We supplemented the qualitative inference from the plot by quantitatively assessing the similarity of distribution using the Krukal–Wallis test. The outcome from Krukal–Wallis test was statistically significant (p-value $< 2.78e-03$) for all metrics (including Churn) when separated by domain and language. The outcome indicates that the distribution of at least one sample of metrics between the three domains and two languages is *different* from the others. To understand the exact nature of the difference(s), we ran pairwise Mann–Whitney–Wilcoxon tests using Cliff's $\delta$ to assess the magnitude of difference. The outcome from the pairwise test for churn and collaboration metrics is shown in Table 5.4. The effect size

[7] We chose churn and collaboration as exemplars to present our observations

Table 5.4: Magnitude of difference in the distribution of churn and collaboration metrics separated by two contextual dimensions (domain and language) and between projects within a domain

| **Dimension** | **X** | **Y** | **Cliffs $\delta$** | |
|---|---|---|---|---|
| | | | **Churn** | **Collaboration** |
| Domain | BR | OS | 0.2148 (S) | 0.1881 (S) |
| | BR | AS | 0.1928 (S) | 0.3062 (S) |
| | OS | AS | 0.0667 (N) | 0.3110 (S) |
| Language | C/C++ | Java | 0.1497 (S) | 0.3078 (S) |
| Project | Chromium | Firefox | 0.0610 (N) | 0.1043 (N) |
| | Linux | OpenBSD | 0.2056 (S) | 0.9915 (L) |
| | Tomcat | WildFly | 0.1153 (N) | 0.9955 (L) |

**Legend**

BR - Browser, OS - Operating System, and AS - Application Server

**Effect Size**

(N) $\delta < 0.147$ (S) $0.147 \leq \delta < 0.33$ (L) $\delta > 0.474$

further provides credence to our inference from the plot that churn is similarly distributed across domains and languages and between projects within a domain. Collaboration, on the other hand, is similarly distributed across domains and languages but differently distributed between projects within the operating system and application server domains.

We summarize the observations from the statistical analyses in Table 5.5. We found that, with the exception of the collaboration metric, all metrics are generalizable.

All metrics, except collaboration, are generalizable (i.e. have similar distributions) across the projects considered in our study irrespective of domain and language.

### 5.3.4 RQ 4 - Thresholds

Question: *Are thresholds of vulnerability discovery metrics effective at classifying risk from vulnerabilities?*

**Motivation.** The benefit of using metrics to support software development is the ability to make objective decisions based on quantifiable aspects of product, process, or people. However, for the metrics to be an effective tool to support decision making, there must be an objective way of saying when the value of a metric indicates specific scenarios. An example of this is the empirical study that found 200 lines of code per hour to be a threshold for individual reviews beyond which there may be degradation in defect discovery [53].

The motivation for the thresholds research question is to leverage existing approaches to compute thresholds for the generalizable vulnerability discovery metrics considered in our

Table 5.5: Summary of the assessment of generalizability of the metrics with ✓ indicating that a metric was found to have a similar distribution with negligible to medium effect size

| **Dimension** | **Churn** | **Collaboration** | **Complexity** | **Contribution** | **Nesting** | **# Inputs** | **# Outputs** | **# Paths** | **Source LOC** |
|---|---|---|---|---|---|---|---|---|---|
| Domain | ✓ | ✓ | ✓ | ✓ | ✓* | ✓ | ✓ | ✓ | ✓ |
| Language | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Project | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓* | ✓ | ✓ |
| **Summary** | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

*Mann–Whitney–Wilcoxon p-value $> 7.94e-04$

study. While we assess the effectiveness of the thresholds to cover historically vulnerable files, our intention is to use the thresholds as triggers to determine if developers should be shown certain feedback about their change.

**Approach.** We used the methodology proposed by Alves *et al.* [5] to compute the thresholds using metric data collected from all six projects considered in our study. The threshold value for the metrics are chosen at the same quantiles (70%, 80%, and 90%) as that in the paper by Alves *et al.* with the risk levels for a metric ($m$) being low ($m < 70\%$), medium ($70\% \leq m < 80\%$), high ($80\% \leq m < 90\%$), and critical ($m \geq 90\%$). We assess the effectiveness of the thresholds in two ways, both of which make use of the historical vulnerabilities data from the Chromium project. Firstly, we quantify the distribution of the historically vulnerable files across the risk levels to ascertain the percentage of vulnerable files that each risk level captures. Secondly, we compute the odds of discovering an historically vulnerable file in each of the risk level. We also compute the odds ratio to assess the change in odds as we move from one risk level to the next.

**Observations.** In addressing RQ 3, we found the collaboration metric to not meet our criteria to be considered generalizable. Therefore, we do not include it in our analysis for RQ 4. Shown in Table 5.6 are the threshold values computed for each of the eight generalizable metrics. By themselves, the metric threshold values provide limited (if any) insights; their utility is to help segregate files into disjoint groups to highlight risk. The risk levels of medium, high, and critical are non-trivial and thus we do not consider the risk level of low in the remainder of this section.

Shown in Table 5.7 is the percentage of historically vulnerable files (offenders) in Chromium covered by each of the three non-trivial risk levels. The percentages indicate the effectiveness of the thresholds, which, on average, captured 23.85% of vulnerable files, going as high as 69.85%, in aggregate, for the contribution metric.

While the percentage of historical files covered by the risk levels is interesting, characterizing the effect of a file moving from one risk level to another is essential if we need to be

Table 5.6: Threshold value of the eight generalizable vulnerability discovery metrics

| **Metric** | **Quantile** | | |
|---|---|---|---|
| | **70%** | **80%** | **90%** |
| Churn | 3,403 | 5,682 | 12,005 |
| Complexity | 197 | 336 | 710 |
| Contribution | 5.53E+04 | 1.59E+05 | 4.95E+05 |
| Nesting | 4 | 5 | 6 |
| # Inputs | 256 | 412 | 865 |
| # Outputs | 261 | 415 | 787 |
| # Paths | 4.15E+03 | 7.86E+04 | 6.97E+07 |
| Source LOC | 1,099 | 1,826 | 3,695 |

able to use thresholds to support any sort of decision making. Shown in Table 5.8 is the odds of discovering a vulnerable file (from the Chromium project) in each of the three non-trivial risk levels. The odds ratios shown in the table quantify the increase in odds of discovering vulnerable files as a file moves from one risk level to next. We also present the ratio of odds in a risk level to that in low. By interpreting the odds ratios we can make inferences such as a file being 14 times more likely to be vulnerable when it moves from low to medium on the churn threshold scale (Odds $\text{Ratio}_{\text{Low}} = 13.9862$).

Table 5.7: Percentage of vulnerable files covered by each of the three non-trivial risk levels (Medium, High, and Critical)

| **Metric** | **% Vulnerable Files Covered** | | | |
|---|---|---|---|---|
| | **Medium** | **High** | **Critical** | **Aggregate** |
| Churn | 7.68% | 8.39% | 5.85% | 21.92% |
| Complexity | 6.62% | 5.96% | 4.64% | 17.22% |
| Contribution | 15.90% | 18.95% | 35.01% | 69.85% |
| Nesting | 9.24% | 5.95% | 4.27% | 19.47% |
| # Inputs | 4.97% | 4.97% | 4.30% | 14.24% |
| # Outputs | 7.28% | 6.62% | 6.62% | 20.53% |
| # Paths | 5.30% | 5.96% | 4.97% | 16.23% |
| Source LOC | 6.26% | 3.44% | 1.68% | 11.37% |
| **Average** | 7.91% | 7.53% | 8.42% | 23.85% |

Table 5.8: Odds of discovering a vulnerable file in each of the three non-trivial risk levels (Medium, High, and Critical) with $\text{Ratio}_x$ being the ratio of odds in a particular risk level to odds in risk level x

| **Metric** | **Medium** | | **High** | | | **Critical** | | |
|---|---|---|---|---|---|---|---|---|
| | **Odds** | **$\text{Ratio}_{\text{Low}}$** | **Odds** | **$\text{Ratio}_{\text{Low}}$** | **$\text{Ratio}_{\text{Medium}}$** | **Odds** | **$\text{Ratio}_{\text{Low}}$** | **$\text{Ratio}_{\text{High}}$** |
| Churn | 4.57E-02 | 13.9862 | 7.96E-02 | 24.3699 | 1.7424 | 1.12E-01 | 34.3548 | 1.4097 |
| Complexity | 9.48E-02 | 4.2210 | 1.61E-01 | 7.1569 | 1.6955 | 2.86E-01 | 12.7234 | 1.7778 |
| Contribution | 1.84E-02 | 3.8177 | 3.52E-02 | 7.3282 | 1.9195 | 1.16E-01 | 24.1669 | 3.2978 |
| Nesting | 6.36E-02 | 3.8456 | 7.39E-02 | 4.4739 | 1.1634 | 8.92E-02 | 5.3960 | 1.2061 |
| # Inputs | 7.65E-02 | 3.2952 | 1.32E-01 | 5.6655 | 1.7193 | 1.97E-01 | 8.4811 | 1.4970 |
| # Outputs | 7.10E-02 | 3.2246 | 8.70E-02 | 3.9511 | 1.2253 | 2.41E-01 | 10.9488 | 2.7711 |
| # Paths | 6.84E-02 | 2.9648 | 9.73E-02 | 4.2188 | 1.4230 | 1.29E-01 | 5.6069 | 1.3290 |
| Source LOC | 1.89E-01 | 10.8511 | 1.93E-01 | 11.0919 | 1.0222 | 2.68E-01 | 15.4085 | 1.3892 |

On average, non-trivial risk levels delineated by thresholds of generalizable vulnerability discovery metrics captured 23.85% of the historically vulnerable files in Chromium, providing support for the effectiveness of the thresholds in classifying risk from vulnerabilities.

## 5.4 Limitations

Generalizability is a concept that is much broader than demonstrating similar distributions. However, in using similarity in distribution as a necessary condition for metric generalizability, we can reason about the potential for statistically-derived thresholds from one project to translate to another. One of the steps in our research approach (an overview of which was provide in Chapter 1) is to leverage developer-in-the-loop to help continually improve our feedback technique and we hope this loop will help strengthen the evidence of generalizability as developers provide their comments on the threshold-driven feedback.

Although we chose projects spanning three application domains and two programming language, we may need to consider more projects to be able to apply our technique at a broader scale. Some of the projects not represented in our sample are small projects with short histories, projects developed in interpreted languages, and closed-source (proprietary) project. Fortunately, most of our analysis is unsupervised and we hope can be easily applied to additional projects.

## 5.5 Summary

In this study, we presented the first step in our approach toward achieving our research vision. We systematically reviewed vulnerability discovery metrics literature and identified a plethora of metrics defined at various levels of granularity. We observed certain limitations in the way these metrics have been validated in the literature, namely: (1) the subjects of study used in the empirical validation tended to be biased to open-source projects, (2) the validation criteria used to reason about the decision-informing ability of the metrics tended to be biased toward assessing association, discrimination, and prediction with little-to-no attention given to actionability and causality, and (3) developers' perception of the metrics was seldom sought in validating the metrics. We implemented and collected ten vulnerability discovery metrics from six open-source projects and assessed their generalizability in terms of similarity in distribution and computed thresholds for the discrete- and continuous-valued metrics using an unsupervised technique proposed by Alves *et al.* [5]. With the exception of one, all discrete- and continuous-valued metrics satisfied our criteria for generalizability. We also found the thresholds to be effective at classifying risk from historically vulnerable files in the Chromium project. The work described in this study forms for the basis for the development of an automated vulnerability discovery interpretation technique to provide developers valuable security insights as they develop software.

# Chapter 6

# Feedback

Software metrics, as Fenton and Neil [32] suggest, have the ability to tell a story to assist developers and managers discover systemic problems in product, process, and people. In a systematic literature review of the vulnerability discovery metrics, we observed that the ability of metrics to support decision making was seldom evaluated using actual developers. In this study, we used vulnerability discovery metrics collected from the Chromium project to provide natural language feedback to developers as they contributing changes. We evaluated developers' perception of the feedback using a survey and their expectations from the feedback using an open dialogue. We further assessed the utility of an existing vulnerability discovery approach—static analysis—to highlight risk from vulnerability contributing commits in FFmpeg. We evaluated the ability of static analysis to highlight risk in vulnerability contributing commits and compared it to that of metrics.

## 6.1 Motivation

The community of researchers have proposed several hundred security metrics [92], in general, and vulnerability discovery metrics (Chapter 5), in particular. However, their proliferation into practice has been limited [91]. The primary factor inhibiting the adoption of metrics in practice is the perception that vulnerability discovery metrics cannot support decision making. However, as we found in systematically reviewing the vulnerability discovery metric literature in Chapter 5, the decision-informing ability of metrics has seldom been assessed using causal modeling or developers' feedback. A metric is deemed not actionable simply because a prediction model that used the vulnerability discovery metric as an explanatory variable was not as effective at predicting vulnerabilities as one would like.

A goal-driven developer or manager is likely to perceive metrics' inability to predict vulnerabilities as the metrics not being actionable. However, the value of metrics, is in their ability to uncover systemic problems in the product, process, and people. If we were to consider "un-actionable (metric) is not useless" [84] and assess the utility of metrics, treated as agents of feedback rather than mere features in a black box prediction model, we are likely

to uncover potential (or the lack thereof) for the use of metrics in practice. In this study, we explore the utility of metrics as agents of security feedback by providing natural language feedback on security to Chromium developers as they contributed changes to the project. We used the values, (unsupervised) thresholds, and interpretation of the metrics collected from the Chromium project to provide the feedback and leverage a qualitative survey to assess developers' perception of the feedback.

We address the following research questions:

**RQ 5 - Feedback**

How is feedback informed by insights from vulnerability discovery metrics perceived by developers?

**RQ 6 - Expectations**

What are developers' expectations from vulnerability discovery metrics?

**RQ 7 - Effectiveness**

How effective are existing vulnerability discovery approaches?

**RQ 8 - Utility**

Is there a utility for vulnerability discovery metrics?

## 6.2 Methodology

In this section, we describe the methodology used to collect and analyze the data to address our research questions. We collected the data from two large open source projects: Chromium, the open-source project behind the Google Chrome web browser which "aims to build a safer, faster, and more stable way for all Internet users to experience the web," and FFmpeg, the "leading [open-source] multimedia framework, able to decode, encode, transcode, mux, demux, stream, filter and play pretty much anything that humans and machines have created." We choose these projects because they were large, popular, and open-source with substantial development history and curated history of vulnerability fixes. We refer to Chromium and FFmpeg as the *subjects of study* in the remainder of this study. We address RQ 5 and RQ 6 in the context of the Chromium project and RQ 7 and RQ 8 in the context of the FFmpeg project. The rationale behind distribution of subjects of study to research questions will become apparent as we describe the data collection and data analysis methodology.

### 6.2.1 Data Collection

In addition to the vulnerability discovery metrics collected from the subjects of study, there are several other pieces of data essential to addressing the research questions. In the subsections that follow, we introduce these peices of data and describe the approach used in collecting them.

**Vulnerability Fixing Commits**

A vulnerability fixing commit is, as the name suggests, a commit that introduced changes to the source code of a project to resolve a vulnerability. The vulnerability fixing commits, and associated metadata, are essential as they identify the timestamp representing the point

in the history of the project when the vulnerability was resolved, the developer responsible for resolving the vulnerability, the type of change (addition, deletion, and/or modification of lines) that resolved the vulnerability, and, more importantly, the file(s) that was(were) modified to resolve the vulnerability. For instance, from commit `546556`[1] in the FFmpeg project, which was reported to resolve the vulnerability identified by CVE-2019-11338, we know that `libavcodec/hevcdec.c` was the vulnerable file and that the vulnerability, which was `Found-by: continuous fuzzing process ...`, was fixed on *March 23, 2019* by *Michael Niedermayer* by adding ten lines of code and deleting four lines of code. Depending on the project, the commit message associated with the vulnerability fixing commit may include additional pieces of information such as bug identifier, code review identifier, and/or the way in which the vulnerability was discovered. For instance, from the commit message associated with commit `ff05b41` in the Chromium project, which was reported to resolve the vulnerability identified by CVE-2016-5216, we know that the commit resolved a bug identified by 653090 (`BUG=chromium:653090`) and the resolution was reviewed in a code review identified by 2418533002 (`Review-Url: https://codereview.chromium.org/2418533002`).

The approach to accurately identify vulnerability fixing commits is predominantly manual with any automation used only to support the manual process. Sabetta and Bezzi [125] attempted to automate the approach to identify vulnerability fixing commits (or, security-relevant commits, as Sabetta and Bezzi referred to them) by proposing a machine learning model that predicted a commit as fixing a vulnerability or otherwise. However, the reported performance (80% precision and 43% recall) of the machine learning model is unacceptable in the context of our work given the downstream impact of incorrectly identified vulnerability fixing commits.

In our study, we collected vulnerability fixing commits for the Chromium and FFmpeg projects by manually perusing the various avenues in which these projects responsibly disclose vulnerabilities. We describe the nuances of, and challenges in, manually identifying vulnerability fixing commits in Chromium and FFmpeg projects below. Since the implementation of the (known) offender metric relies on the ability to identify vulnerability fixing commits, the effort expelled in identifying the vulnerability fixing commits is serving the dual purpose of providing data to the (known) offender metric and to the addressing of the research questions.

**Chromium** The Chromium project publishes a list of all security fixes implemented in a particular release in the blog post on Chrome Releases Blog[2] that announces the release. Each security fix includes a Common Vulnerabilities and Exposures identifier and an identifier to an entry in the bug tracking system that describes the vulnerability. The bug identifier can in turn be used to identify the commit that resolved the vulnerability by relying on the knowledge that the Chromium project follows a practice of including bug identifiers in the message of a commit (if the commit resolved a bug). As we curated vulnerability fixes from the Chrome Releases Blog, we noticed that some publicly disclosed vulnerabilities were not mentioned in any of the blog posts. We overcame this limitation by supplementing the vulnerability fixes from the Chrome Releases Blog with those from Monorail[3], the bug tracking system that the Chromium project uses, and by manually reviewing the vulnerability reports associated with `cpe:/a:google:chrome` (the Common Platform Enumerations identifier assigned to Google Chrome) on the National Vulnerability Database [1]. In total, we identified 1,453 vulnerabilities in the Chromium project that were associated with a bug identifier. 1,079 of these vulnerabilities were identified from Monorail, 105 from the Chrome Releases Blog, and 269 by manual review of the National Vulnerability Database.

[1] `https://github.com/ffmpeg/ffmpeg/commit/546556` [2] `https://chromereleases.googleblog.com/`
[3] `https://bugs.chromium.org/p/chromium/issues`

**FFmpeg** The FFmpeg project publishes the Common Vulnerabilities and Exposures identifiers of all vulnerabilities that have been resolved in the project on the Security Section[4] of their project website. The page conveniently includes the identifier of the commit that resolved the vulnerability. However, fixes to some of the earlier vulnerabilities do not include the commit identifier. We reviewed the vulnerability reports of such vulnerabilities on the National Vulnerability Database [1] which sometimes included a link to the fix in their source code repository. In cases where the vulnerability reports did not have a reference to the fix, we searched the repository for the Common Vulnerabilities and Exposures identifier. In total, we identified 341 commits that were reported to have fixed 287 vulnerabilities in the FFmpeg project. 96.77% (330) of these commits fixed a single vulnerability, 2.93% (10) fixed two vulnerabilities, and 0.29% (1) fixed three vulnerabilities. Furthermore, 82.23% (236) of the vulnerabilities were fixed in a single attempt (i.e. one commit was attributed to have fixed the vulnerability), 14.29% (41) of the vulnerabilities took two attempts, 2.44% (7) took three attempts, 0.70% (2) took four attempts, and 0.35% (1) took six attempts. The 341 vulnerability fixing commits modified a total of 184 files. Although a considerable portion (63.04%) of these (vulnerable) files were fixed for a single vulnerability, three files (`libavcodec/utils.c`, `libavcodec/h264.c`, and `libavcodec/jpeg2000dec.c`) were fixed for eight vulnerabilities lending further credence to the utility of the (known) offender metric [85].

Shown in Figure 6.1 is the distribution of churn by type (insertion or deletion) in vulnerability fixing commits in the FFmpeg project. As can be inferred from Figure 6.1, vulnerability fixing commits tend to insert lines than delete lines to fix vulnerabilities. We found quantitative evidence to support this inference leading us to conclude that vulnerability fixing commits in the FFmpeg project are statistically (Mann-Whitney-Wilcoxon p-value $\ll 0.01$) and practically (Cliff's $\delta = 0.5419$ representing a large effect) more likely to insert lines than delete lines.

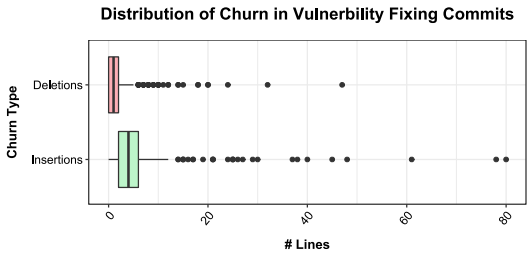



Figure 6.1: Distribution of churn in vulnerability fixing commits in the FFmpeg project

**Vulnerability Contributing Commits**

The vulnerability fixing commits are essential to characterizing vulnerabilities in software as they identify files that were definitively vulnerable prior to the fix. However, any attempt to characterize security vulnerabilities using attributes of software at the time of vulnerability fix is unlikely to yield insights that could foreshadow vulnerabilities because the attributes are describing if a file is likely to *need a fix for a vulnerability* rather than if

[4] `http://ffmpeg.org/security.html`

a file is likely to *be vulnerable*. The difference is subtle but its implications to the practical application of the insights could be profound. The temporality of vulnerabilities is the key contributor to this disparity. As Meneely et al. [85] report, vulnerabilities tend to exist in software for many years before they are exploited by attackers or discovered by developers. The attributes of software at the time of the *fixing* of the vulnerability may not be the same as that at the time of the *introduction* of the vulnerability. The key, therefore, is to characterize vulnerabilities when they are introduced rather than when they are fixed. However, while vulnerability fixing commits are disclosed when a vulnerability is fixed, commits that introduced the vulnerabilities, referred to as *vulnerability contributing commits* [85, 115], are seldom disclosed.

In our work, one of the lines of inquiry is to assess the effectiveness of existing vulnerability discovery approaches in highlighting risk of vulnerability from changes to software as the developers introduce them. As a result, identifying vulnerability contributing commits is no longer a preference but a necessity. In their work, Meneely et al. [85] manually identified 124 vulnerability contributing commits that could be attributed to have contributed the 68 vulnerabilities in Apache HTTP. In principle, the manual approach to identifying vulnerability contributing commits, involving the two steps enumerated below, is quite simple.
For each file changed in a vulnerability fixing commit,

Step 1: Identify suspect lines by perusing the change that fixed a vulnerability. Suspect lines can be of two types: (1) lines modified to fix the vulnerability and (2) lines before and after (commonly referred to as context lines) the lines inserted to fix the vulnerability.

The intuition is that lines modified to fix a vulnerability are likely the lines that led to the vulnerability and the lines before and after the lines inserted to fix the vulnerability represent the snippet in which code that could likely have prevented the vulnerability was missed.

Step 2: Identify the commit that last modified each of the suspect lines.

The intuition is that the commit that last modified the line that led to the vulnerability or the line near which the code to prevent the vulnerability was missed is attributable as having contributed to the vulnerability.

The seemingly simple approach can be quite tricky in practice because the step to identify the commit that last modified a suspect line is accomplished using features provided by the system used to manage the repository containing the project source code. For instance, if source code repository was managed by `git`, one could use `git blame -L n,m fix^ -- path` to identify the last commit that modified each line in the interval $[n, m]$ in the file identified by `path` prior to it being fixed for a vulnerability (specified as `fix^`, where `^` represents the parent of the commit identified by `fix`).

The manual approach to identifying vulnerability contributing commits used by Meneely et al. [85] is not scalable, especially when we have 341 vulnerability fixing commits in the FFmpeg project. The alternative is to leverage the SZZ Algorithm [140], named after the researchers who proposed it (Śliwerski, Zimmermann, and Zeller), to trace vulnerability fixing commits to vulnerability contributing commits. However, as Rodríguez-Pérez et al. [123] mention, publicly-available implementations of the SZZ Algorithm are few and far apart. Borg et al. [12] attempted to mitigate this limitation by developing SZZ Unleashed,[5] an open implementation of the SZZ Algorithm for `git` repositories.

[5] `https://github.com/wogscpar/SZZUnleashed`

While the availability of SZZ Unleashed is a boon to the research community, its (implicit) coupling with an issue tracker (Atlassian Jira) limits its applicability to projects that do not actively maintain an issue tracker system. We worked around this limitation by skipping steps in SZZ Unleashed that were dependent on the issue tracker. However, SZZ Unleashed failed to identify vulnerability contributing commit for 110 (32.26%) of the 341 vulnerability fixing commits in FFmpeg. We presumed that this discrepancy was likely an implication of skipping steps that were dependent on the issue tracker. As an alternative, we resorted to using the heuristics-based approach to identify vulnerability contributing commits used by Perl et al. [115].

The approach used by Perl et al. [115], being similar to that used by Meneely et al. [85], is intuitive. We were unable to find an open implementation of the approach used by Perl et al. [115] so we implemented the approach ourselves. We have released the implementation as an open-source utility named `archeogit` available on GitHub at `https://github.com/samaritan/archeogit`.

We used `archeogit` [98] to identify 309 vulnerability contributing commits in FFmpeg which contributed to one or more vulnerabilities which were later resolved by one (or more) of the 341 vulnerability fixing commits identified earlier. 83.82% (259) commits were found to have contributed code to a single vulnerability, 11.97% (37) to two vulnerabilities, 1.94% (6) to three and five vulnerabilities, and 0.32% (1) to four vulnerabilities.

Since `archeogit` is an utility developed to support this research, we verified and validated it using a random sample of 205[6] vulnerability contributing commits. The verification (Does `archeogit` produce expected results?) did not uncover any flaws in the implementation but the validation (Do the vulnerability contributing commits reported by `archeogit` actually contribute the vulnerability?) did uncover potential flaws. The only way to mitigate the potential flaws identified during the validation of `archeogit` is to use a manual approach, which can be a prohibitive endeavor. Fortunately, the Vulnerability History Project (`https://vulnerabilityhistory.org/`), a brainchild of Andrew Meneely, is working to address just this need in the community. In the Vulnerability History Project, undergraduate students in the Engineering Secure Software (SWEN-331) course at Rochester Institute of Technology manually curate the history of vulnerabilities in popular open-source project through an established collaborative process.

A vulnerability contributing commit can contribute a vulnerability in one of two ways: the vulnerability is contributed when the file is *added* for the first time or the vulnerability is contributed when the file is *modified*. 73.23% (227) of the vulnerability contributing commits in FFmpeg contributed a vulnerability when modifying a file and the remaining 26.77% (83) contributed a vulnerability when adding a file. There is a mismatch in the number of vulnerability contributing commits here ($310 = 227 + 83$) from that mentioned earlier (309) because one of the vulnerability contributing commits (`07c55d8ea3`) contributed a vulnerability when adding `libavcodec/vorbis_dec.c` and when modifying `libavcodec/vorbis.c`.

**Static Analysis**

Chess and McGraw [19], in advocating for the (regular) use of static analysis to improve software security, suggest that one should "aim for good, not perfect." The quote succinctly highlights both the good and the bad of static analysis. In principle, static analysis techniques look for patterns in source code being analyzed; patterns that represent *known* security problems. The limitation of this approach is that security problems for which no patterns exist yet are unlikely to be detected by static analysis. However, in the spirit of "aiming for good" [19], static analysis can be used as an effective means to prevent mistakes that are

[6] Computed using the SurveyMonkey Sample Size Calculator [144] using 95% as the Confidence Level and 5% as the Margin of Error

known to lead to security problems, providing developers an objective approach to improving the overall security of software. The design philosophy of Splint, a lightweight static analysis tool, proposed by Evans and Larochelle [30] further contributes to the vision of having static analysis be an integrated part of a modern software development workflow.

The potential for leveraging static analysis to improve software security certainly exists, however, its effectiveness in detecting real vulnerabilities has been a subject of inquiry in the research community. In proposing a static analysis technique, Livshits and Lam [68] reported that their technique was effective at identifying potential security vulnerabilities in Java applications with a relatively low false positive rate of 29.27% (12 out of 41). Walden and Doyle [148], in proposing a metric called Static Analysis Vulnerability Indicator, found that the density of vulnerabilities reported by static analysis (Fortify Source Code Analyzer, in their study) was strongly correlated ($\rho = 0.67$) with number of vulnerabilities discovered after release. These studies suggest that static analysis techniques have the potential for detecting actual vulnerabilities.

Over the years, several static analyzers have been developed by academicians, open-source communities, and commercial enterprises. Li and Cui [65] reviewed and compared eight static analyzers available for C, C++, and Java project and concluded that the shortcomings of individual analyzers may be overcome by using multiple analyzers together. However, a challenge in using multiple analyzers together is that the overlap among vulnerabilities reported by the analyzers must be collapsed to make the report more usable. Code Dx,[7] a commercial application vulnerability management tool, aggregates vulnerabilities reported by multiple analyzers to provide a holistic perspective on the security of software.

In our work, we assess the ability of static analyzers to detect vulnerabilities in the code contributed by vulnerability contributing commits. By definition, the vulnerability contributing commits are contributing vulnerabilities and assessing the static analyzers' ability to identify these vulnerabilities highlights their usefulness in assisting developers engineer secure software. Taking note of the suggestion by Li and Cui [65], we use findings[8] reported by three static analyzers: Cppcheck [27], Flawfinder [151], and Rough Auditing Tool for Security (RATS) [128]. While we did translate the severity of findings reported by the three static analyzers to a normalized scale to simplify analysis, we did not collapse the overlap in the reported findings. Shown in Table 6.1 is the translation of severity of findings reported by Cppcheck, Flawfinder, and RATS to the uniform scale. We have six levels in our normalized severity scale with 1 (Trivial) and 6 (Catastrophic) being the lowest and highest severity, respectively. Our choice of six as the number of levels was predicated on the six severity levels that both Cppcheck and Flawfinder use. RATS, which uses a four level severity scale (default, low, medium, and high), was introduced to data collection after the normalized severity scale was developed and, as a result, we have two levels in the normalized severity scale that do not translate to a severity reported by RATS.

We used the High Performance Computing Cluster [122] at the Rochester Institute of Technology (RIT) to parallelize the static analysis of multiple versions of a subject of study using Cppcheck, Flawfinder, and RATS.

### 6.2.2 Data Analysis

In the subsections that follow, we describe the approach used to analyze the data collected from the subjects of study to address the research questions. Each subsection concludes with

[7] `https://codedx.com` [8] We use the term *finding* (instead of vulnerability) to refer to static-analyzer-reported vulnerability to avoid confusion with actual vulnerabilities.

Table 6.1: Translation of severity of findings reported by Cppcheck, Flawfinder, and RATS to a normalized severity scale

| Cppcheck | Flawfinder | RATS* | Normalized |
|---|---|---|---|
| information | 0 | Default | Trivial |
| portability | 1 | - | Low |
| performance | 2 | - | Medium |
| style | 3 | Low | High |
| warning | 4 | Medium | Critical |
| error | 5 | High | Catastrophic |

*RATS uses four severity levels so we chose to not translate severity of 2 and 3 on the normalized severity scale to a severity from RATS.

a reference to the relevant research question(s) that the analysis pertains to.

**Computing Metric Thresholds**

Predictability [84] is often the criterion used to reason about the empirical validity of vulnerability discovery metrics. The prerequisite for assessing predictability, however, is the availability of a dataset of historical vulnerabilities to train and test a vulnerability prediction model. While some projects, in the interest of responsible vulnerability disclosure [17], have developed mechanisms for the curation of historical vulnerabilities, satisfying this prerequisite, a model trained with data from one project may not be directly applicable to discover vulnerabilities in a different project [162]. An unsupervised approach is, therefore, needed.

As researchers empirically validate vulnerability discovery metrics using predictability as a criterion, related criteria such as association and discriminative power as often evaluated as well. The outcomes from association and discriminative power often reveal important empirical information about the relationship between vulnerability discovery metrics and historical vulnerabilities. The polarity of the association (if one exists) between vulnerability discovery metrics and historical vulnerabilities is one such key piece empirical information that is likely to be transferable from one project to another. For instance, Zimmermann et al. [160] found that total (cyclomatic) complexity was positively correlated with number of vulnerabilities in Windows Vista binaries and Shin et al. [133] found a positive association between total (cyclomatic) complexity and vulnerabilities in files in both Mozilla Firefox and Red Hat Enterprise Linux. In both these studies, high total complexity was associated with vulnerabilities, however, the missing piece of information is the threshold value at which total complexity indicates a problem.

If we had acceptable thresholds for the vulnerability discovery metrics, we could apply the thresholds to label an entity being measured to exhibit certain attribute to a higher or lower degree. While computing a universal threshold for all projects is intractable, Lanza and Marinescu suggest that statistical information rankings may be used to determine explicable thresholds [61]. In our study, we used the methodology proposed by Alves et al. [5] to

compute the thresholds. Despite being an unsupervised, the methodology proposed by Alves et al. [5] was found to be effective (1) in predicting fault-proneness when compared with other supervised approaches [14] and (2) at being able to delineate risk levels that captured, on average, about one-fourth of the historically vulnerable files in Chromium[102].

As proposed by Alves et al. [5], we compute the thresholds by taking metric values collected from a projects and using the metric values at 70%, 80%, and 90% (the quantiles proposed by [5]) of the metric value distribution to delineate the risk levels. We use the following levels to assess risk using a metric ($m$): low ($m < 70\%$), medium ($70\% \leq m < 80\%$), high ($80\% \leq m < 90\%$), and critical ($m \geq 90\%$). The approach to compute thresholds, being based on the distribution of metric values, cannot be applied to non-numeric metrics like (known) offender. Since we know the (known) offender is a boolean-valued metric, we consider a file to be risky if it is a known offender.

To simplify aggregation of risk assessed using multiple metrics, we assign numerical values of 0, 1, 2, and 3 to the low, medium, high, and critical, respectively. We used the source lines of code of a file to normalize the other metrics collected from the file since source lines of code in a file has shown to be correlated with typical metrics collected from a file (See work by Jay et al. [50] for an investigation of the linear relationship between source lines of code and cyclomatic complexity). We do not normalize the metrics collected at the change level, commit level, and developer level because we found no strong inter correlation among the metrics collected at these levels of granularity.

The implicit assumption in the approach to computing thresholds proposed by Alves et al. [5] is that higher values of a metric indicate a concern of interest (vulnerabilities, in our case). However, there are vulnerability discovery metrics (ownership evaluated by Perl et al. [115], for instance) for which lower values indicate a concern of interest. In case of such metrics, we consider the mirror of the distribution of metric values when computing the thresholds. In case of ownership, we simply subtract the metric values from 1.0 to get the mirror distribution. For instance, if the ownership value of a developer is 0.37 (i.e. 37% of the commits to the project were made by the developer), then 0.73 = 1.00 - 0.37 represents the mirror value (i.e. 73% of the commits to the project were made by other developers).

In addressing the feedback research question (RQ 5), we use the metrics' thresholds, coupled with corresponding metrics' values, to generate natural language feedback on security and to determine when to present the security feedback to developers. In addressing the utility research question (RQ 8), we use the metrics' thresholds to classify risk from changes in vulnerability contributing commits, the commit itself, the file(s) in the commit, and developer contributing the commit. We do not use the same version of a project to compute the metrics' thresholds and use the thresholds to classify risk. The rationale for this decision is to simulate a likely scenario in which thresholds computed in the past are used to highlight risk from future changes.

## 6.3 Results

In the subsections that follow, we present our empirical analysis and observations to address the research questions.

### 6.3.1 RQ 5 - Feedback

Question: *How is feedback informed by insights from vulnerability discovery metrics perceived by developers?*

**Motivation.** Although the research literature is rife with security metrics [92], their adoption in practice to assess security, in general, and to discover vulnerabilities, in particular, has been limited. The subpar performance of the prediction models that use vulnerability discovery metrics as features and the granularity at which these models predict vulnerabilities are cited as inhibitors to the use of vulnerability discovery metrics in practice [91]. The reliance on performance measures (usually precision, recall, and F-measure) of (black box) vulnerability prediction models to assess the utility of vulnerability discovery metrics, though objective, has several limitations, chief among which are:

1. A prediction from a black box vulnerability prediction model is seldom accompanied by context to enable a developer to interpret the prediction. For instance, if a black box vulnerability prediction model was to tell developers that "`foo.c` is likely, with certain probability, to be vulnerable to exploit," the first question a developer is likely to ask is "Why?". If, on the other hand, we were to interpret the vulnerability discovery metrics that were used as features in the model to tell developers that "`foo.c` is likely, with certain probability, to be vulnerable to exploit because it has been modified a lot and has been vulnerable in the past.",[9] the developers can use the accompanying context to decide whether or not to investigate further.
2. A metric that likely has the potential to assist developers in improving the overall security of software may be unduly regarded as non-utilitarian simply because it does not improve the performance of a state-of-the-art vulnerability prediction model. For instance, the proximity to the attack surface metric [100] may not improve the performance of a vulnerability prediction model built with traditional metrics [160] but the insights from the proximity to the attack surface metric may be perceived as more relevant by a developer than that from the traditional metrics.

As a consequence of the aforementioned limitations, vulnerability discovery metrics are seldom seen as an useful approach to assess the overall security of software, in general, and to discover vulnerabilities in software, in particular. The key to encouraging adoption of vulnerability discovery metrics in practice is to think of these metrics not as mere features in a black box prediction model but as agents of (natural language) feedback.

The motivation for the feedback research question is to assess developers' perception of feedback informed by insights from vulnerability discovery metrics. While the ideal is for the feedback to lead to the discovery of a vulnerability, we hypothesize that developers can still benefit from being made aware of the engineering failures known to be associated with historical vulnerabilities. The increased awareness may contribute to the developers' security mindset which, as Bruce Schneier argues [129], is increasingly becoming an essential skill for security professionals, in general, and software engineers, in particular.

**Approach.** We address the feedback research question by providing developers natural language security feedback informed by insights from vulnerability discovery metrics and assessing the developers' perception of the feedback. We address this research question using a subset of eight (line churn, collaboration, complexity, contribution, nesting, # inputs, # outputs, and # paths) of the eighteen metrics described in Appendix A.1. Furthermore, we address this research question in the context of the Chromium project alone, the reasoning for which will soon become apparent.

The two steps involved in addressing the feedback research question were (1) to provide the feedback to Chromium developers and (2) to assess their perception of the feedback.

[9] The two metrics used in this hypothetical scenario are line churn [160] and (known) offender [85], respectively.

*Step 1: Provide Feedback*

We had to address three key concerns to provide feedback to Chromium developers: (1) Where should the feedback be provided?, (2) When should the feedback be provided?, and (3) What should the feedback contain?.

*Where should the feedback be provided?* Developers tend to have specific expectations from the tools that they use when engineering software [62, 52, 34] and the expectation that the tool not interrupt their existing workflows is paramount [52]. Code review, being a common practice in many mature software engineering workflows today, provides an ideal opportunity to provide feedback on security [7]. Tricoder, a program analysis ecosystem at Google, does just this but with static analysis warnings [126]. We chose to use code reviews as a medium to provide the natural language feedback on security addressing the concern on where the feedback should be provided. The use of code reviews has the added benefit of providing developers participating in the review an opportunity to have a conversation about the security feedback aiding knowledge transfer and enabling us to leverage a developer-in-the-loop approach to improving the feedback.

We chose the Chromium project because all changes to the source code are *required* to be reviewed [23] which implies that code review is already a part of Chromium developers' workflow. Furthermore, the Chromium project has been at the forefront of introducing practices to improve security of software through initiatives like the Chrome Vulnerability Reward Program[10] and ClusterFuzz.[11] The Chromium project uses Gerrit,[12] an open-source code review system, to facilitate the code review process. We provide the security feedback as a comment on open code reviews in the Chromium project.

*When should the feedback be provided?* The act of providing developers security feedback is akin to an intervention. The developer receiving the security feedback must read, consider, and (possibly) discuss the feedback with other developers to determine whether to act on the feedback or not. Therefore, the concern of when to provide the feedback is crucial to avoid being labelled as the "the boy who cried wolf" [2]. A trivial solution like using the predicted probability that a file is likely to need a fix for a vulnerability from a prediction model and provide the feedback when the probability is higher than a threshold (say, 0.75) is infeasible owing to the prerequisite for dataset with which to train the prediction model.

In our work, we used the methodology proposed by Alves et al. [5] to associate metric values ($v$) with a weight and mapping the weight using prescribed quantiles of 70%, 80%, and 90% to risk levels using the scale low ($v < 70\%$), medium ($70\% \leq v < 80\%$), high ($80\% \leq v < 90\%$), and critical ($v \geq 90\%$). We collected the metrics from Chromium at `6b9bf768231f` and associated each metric value with weights enabling us to determine the threshold values for each metric. As mentioned earlier, we chose to provide security feedback as comments on open code reviews in the Chromium project. As a result, we had to first identify the open code reviews and the files that were being reviewed. We developed a utility to mine Gerrit using its RESTful API for open code reviews. We then mapped each file in each open code review to the metric values and the corresponding weights. We averaged the individual metric weights to assign each file a weight and then averaged the file weights to assign each code review a weight. We then picked one or more of the open code reviews with the highest weight as the candidates to provide the feedback on, addressing the concern of when should the feedback be provided.

We automated the entire workflow such that at 12:00 AM each day Gerrit was mined for open code reviews and each open code review was assigned a weight. The Gerrit RESTful

[10] `https://www.google.com/about/appsecurity/chrome-rewards/index.html`

[11] `https://google.github.io/clusterfuzz/` [12] `https://www.gerritcodereview.com/`

API has an upper limit on the number of code reviews that it returns and the limit was 1,000 when we mined the code reviews. Each morning, we had a fresh set of 1,000 candidate code reviews to review and provide feedback on. We tracked all code reviews that we had provided a feedback on. We also tracked the developers and reviewers who participated in code reviews that we had provided a feedback on. When reviewing candidate code reviews to provide feedback on, to get as varied a perception as possible we made sure the developer and reviewers participating in the review were not already provided feedback on in a past code review.

*What should the feedback contain?* In their work proposing interpretable decision sets, Lakkaraju et al. [59] state that "Interpretable models ... bridge the gap between domain experts and data scientists.". We, as researchers, should work toward translating our metrics to interpretable and actionable insights for developers. As with addressing the concern of when to provide the feedback, we could train an interpretable machine learning model[13] like Logistic Regression or Decision Trees and use their interpretation in the feedback. However, the lack of historical vulnerabilities and the cross-project incompatibility of the models render the trivial solution infeasible. Furthermore, the models, though interpretable, may not be directly comprehensible to the developer because the interpretation of the models is more than just metric coefficients (in Logistic Regression) or split cutoffs (in Decision Trees). As researchers propose metrics, they painstakingly theorize the attributes that the metric quantifies and then assess its validity to quantify the attribute and its relationship with a quality attribute of software (vulnerability, in our case). The feedback that we provide developers must interpret the metric in the context of the attribute that the metric quantifies. The developers are more likely to be aware of these attributes as they are related to the domain of Software Engineering. For instance, the collaboration centrality metric [81] may not be directly comprehensible by developers, however, the attribute that it aims to quantify—structure of developer collaboration—is more amenable to developer comprehension.

In our work, we used the weight associated with a metric value, determined using the approach proposed by Alves et al. [5], and the risk level that it represents to determine the metrics that are relevant for a given file. As mentioned earlier, the open code reviews with the highest weight were considered as candidates for providing security feedback on. For each such candidate code review, we selected all metrics whose values ($v$) indicated at least a medium risk (using the scale $70\% \leq v < 80\%$ where 70% and 80% are the threshold values for the metric in question). We then manually designed the feedback which included a list of file paths, from the code review in question, and a natural language description of the risk associated with the file. The natural language description included a narrative describing the attributes that each risk-indicating metric measures with the metric value and the corresponding weight used to support the narrative.

*Step 2: Assess Perception*

We assessed developers' perception of the security feedback through an online anonymous survey the link to which was included in the feedback that we provided. The survey had three questions, the first of which elicited developers' perception of the feedback via eight five-level Likert scale (levels from strongly disagree to strongly agree) statements, the second elicited developers' experience in the Chromium project and in software development, and the third elicited anything else the developers wanted to communicate. The questions, and associated statements (if any), from the survey are included in Appendix H in the Appendix to this paper.

Since the assessment involves human subjects (Chromium developers), the survey was

[13] See Interpretable Machine Learning by Molnar [90] for an overview

Table 6.2: Threshold values for eight metrics collected from the Chromium project at `6b9bf768231f`

| Metric | Quantiles | | |
|---|---|---|---|
| | **70%** | **80%** | **90%** |
| Line Churn | 2,954 | 5,421 | 12,164 |
| Collaboration | 217.40 | 285.94 | 438.66 |
| Complexity | 142 | 266 | 785 |
| Contribution | 394,024.66 | 778,280.86 | 2,207,375.77 |
| Nesting | 3 | 4 | 5 |
| # Inputs | 179 | 326 | 898 |
| # Outputs | 319 | 554 | 1,222 |
| # Paths | 472 | 6,399 | 3,484,322 |

reviewed and approved by the Institutional Review Board at Rochester Institute of Technology. The approval from the review board is shown in Figure G.1 in Appendix G. As mentioned earlier, the survey was anonymous and entirely voluntary. Furthermore, developers who volunteer to participate in the survey are presented with an informed consent form describing the study, in general, and the survey, in particular, in more detail. The developers are provided a link using which they can download a copy of the informed consent form for their record.

**Observations.** We collected the eight vulnerability discovery metrics from the Chromium project at `6b9bf768231f` and, using the approach proposed by Alves et al. [5], associated each metric value with weights enabling us to determine the threshold values for each metric. Shown in Table 6.2 are the threshold values for each of the eight metrics in the Chromium project.

We used the metric values, the weights associated with the metric values, and the aforementioned thresholds to develop the feedback, identify the code review that we were going to provide the feedback on, and post the security feedback as a comment on the identified code review. Shown in Figure I.1 in the Appendix is a sample security feedback provided to Chromium developers in one of the code reviews.

We provided security feedback on 19 code reviews from June 13, 2019 to June 27, 2019, averaging about 2 code reviews per week day. The survey link in the first four instances of the security feedback we provided did not track the code review on which the feedback was provided on making it hard to associate developers' perception of the security feedback to the actual feedback. We modified the survey link to include the unique identifier of the code review on which the security feedback was provided. The inclusion of the unique identifier in the link does not violate the anonymity of the survey because a code review has more than one person involved (developer and one or more reviewers) and the survey can be associated with the code review but not the actual developer providing their perception. Furthermore, the code review and any comments on it are publicly accessible so the survey can conceivably be accessed by anyone with access to the code review comments.

We received a total of five complete survey responses which places the response rate at 26.31% (assuming one response from each of the 19 code reviews we provided a feedback on). The first of the five responses seems like the developer was just trolling us because the developer responded with 0 years on both experience questions (Q2) and in the additional information (Q3) included a link to an animated GIF. We were left with four legitimate responses which, is not a sizeable sample to draw any sort of generalizable conclusions from. After we had provided feedback on nine code reviews, one of the developers participating on a code review we provided feedback on reached out to us via email expressing their recognition of our effort to improve security in Chromium. The developer suggested that code reviews may not be the right medium for the sort of feedback we were providing and that `chromium-dev@chromium.org` and/or `security-dev@chromium.org` Google Groups were more appropriate. After a cursory investigation of both groups referenced, we concluded that these groups were generic and not amenable to the sort of (contextual) feedback we were providing. We responded to the developer and continued to provide security feedback on code reviews. We provided feedback on 11 more code reviews but on July 1, 2019 we noticed that we could no longer access the code reviews that we had provided feedback on when authenticated to the Gerrit web interface. We created an issue with the Gerrit team and were told "You are getting a 404 because you have been explicitly banned from Gerrit on the chromium host for spammy comments you made on code reviews ..."

After a couple of weeks, we modified our approach to use `security-dev@chromium.org` as the medium to provide the feedback on to. We included a link to the code review that the feedback was for to include additional context in the feedback. We posted two message to `security-dev@chromium.org` and got two complete survey responses. In the second response, the developer responded to the additional information question (Q3) by saying "This data is useless and these emails are essentially spam." Fearing being banned from `security-dev@chromium.org` for posting spam, we ceased providing feedback to developers to take a step back and reevaluate our approach.

In total, we got six (seven minus one invalid response) complete survey responses which, as mentioned earlier, is not a sizeable sample to draw any sort of generalizable conclusions from. However, we summarize the responses here for completeness. On average, the participants reported 11.33 and 3.77 years of experience in software development and in Chromium project, respectively. We aggregated responses to the statements assessing developers' perception of the security feedback (Q1) by computing the median of the individual responses (mapped on to a numerical scale spanning from Strongly Agree at 1 to Strongly Disagree at 5). We then rounded the aggregate response to the nearest whole number and mapped the number back to the Likert scale to simplify interpretation.

The developers reported (1) to have carefully read and understood the feedback implying that the feedback was clear, (2) being neutral toward the feedback disrupting their workflow and receiving the feedback in a group setting implying that the choice of code reviews as a medium for providing security feedback is probably appropriate, (3) being neutral about discussing the feedback with their peers implying that the feedback did serve as a catalyst for security conversation, albeit not as consistently as we had hoped for, (4) not being surprised by the feedback implying that the developers may have been aware of the concerns highlighted in the feedback or that concerns raised were expected, (5) the feedback having not encouraged them to think about security. The only unanimous response from all developers was that feedback was not useful.

In three of the six complete survey responses, developers provided additional information (in response to Q3) as free form text. We analyze these responses in addressing the expectations research question.

In aggregate, Chromium developers reported that security feedback informed by insights from vulnerability discovery metrics was, despite being read, understood, and discussed, not surprising, at best, and not useful, at worst.

### 6.3.2 RQ 6 - Expectations

Question: *What are developers' expectations from vulnerability discovery metrics?*

**Motivation.** In addressing the feedback research question, we discovered that Chromium developers did not perceive the feedback informed by insights from vulnerability discovery metrics to be useful in assisting them improve the security of their software, in general, and discover vulnerabilities, in particular. The expectations research question is an opportunistic extension to the feedback research question, in that, the observations from the feedback research question led us to critically inquire the fundamental purpose of metrics, in general, and the Chromium developers' perception of vulnerability discovery metrics and the broader metrics research community, in particular.

The motivation for the expectations research question is to elicit developers' expectations from vulnerability discovery metrics and to assess if the metrics can fulfil these expectations.

**Approach.** We began to address the expectations research question by initiating an open conversation with Chromium developers by posting a message (Shown in Figure J.1 in the Appendix) on the `chromium-dev` Google Group. Our intention was not to collect, and summarize, a sizeable sample of monologue responses (as surveys typically do) from developers but, rather, to connect with the practitioners that the research that we produce is aimed at helping and understand their awareness of the research and its usefulness in their practice.

In the message posted to `chromium-dev` to initiate the conversation (See J.1 in the Appendix), we provided context of the project and a summary of our observations from posting security feedback on code reviews. We included four open-ended and high-level questions (shown below) in the message to serve as a catalyst for the conversation.

1. What are your thoughts on the security metrics research community?
2. Are we producing work that could be valuable to you?
3. Do you think there is a place for security metrics in the Chromium development life cycle?
4. When, in the Chromium development life cycle, do you think is an appropriate opportunity for providing metrics-derived security feedback?

One of the early responses to our message included a suggestion that we post the message to `security-dev` Google Group to "get the attention of the security-focused members of the Chromium community" as "Not everyone keeps up with all of the mail on chromium-dev@." We acted on the suggestion and posted a similar message to `security-dev` to have a more focused conversation.

We used the approach described by Auerbach and Silverstein [6] to analyze the qualitative data that the conversation with the Chromium developers produced. The approach, which has five steps, is bottom up with each step raising the level of abstraction at which we reason about the data. The five steps in the approach involve (1) the extraction of relevant text from the raw responses, (2) the identification of repeating ideas in the relevant text, (3) the grouping of similar repeating ideas into themes, (4) the grouping of similar themes

into more abstract theoretical constructs, and finally, (5) the development of theoretical narratives by organizing the theoretical constructs. In presenting the analysis, we support our interpretation of the responses by including quotations from the developers to be transparent with the subjectivity inherent in qualitative analysis.

**Observations.** We had a total of seven developers respond to our message across both `chromium-dev` and `security-dev`. We had more conversations with developers on `chromium-dev` than on `security-dev` with five of the seven developers responding on `chromium-dev`. The conversations with three of the seven developers was private, in that, the developers choose to respond directly to us without including the Google Group email address. We respected their choice when responding to their messages so the response was not publicly posted to the Google Group. We had only two of the seven developers with whom we exchanged multiple messages.

We gathered all the messages in our conversations, in their raw format, from the `chromium-dev` and `security-dev` Google Groups and our mailboxes into a single location. We applied the approach described by Auerbach and Silverstein [6] to analyze only those messages that we received from the developers since our responses to the developers are not relevant to the research question. We had ten such messages across all seven developers who responded (either publicly or privately).

We extracted relevant text from the responses by removing any quoted text that email clients tend to automatically include when responding to a message. We eliminated one response at this stage because the response was just "iok", which we interpreted as the acronym for *it's okay*, commonly used in chat-based conversations. Although we were familiar with the responses, having replied to most of them, we perused the responses again to familiarize ourselves with the specific expectations and/or concerns that the developers talked about. We then systematically reviewed every response to identify excerpts that expressed an expectation or suggestion. As we aggregated the excerpts from multiple responses, we observed repeating expectations and/or suggestions. We grouped excerpts representing such repeating expectations and/or suggestions into themes. As we labeled the themes, we observed that the expectations and/or suggestions were about the approach we used to provide the security feedback or about the feedback itself. We grouped themes pertaining to the approach we used to provide security feedback and those pertaining to the feedback itself into independent theoretical concepts. The theoretical narratives that follow leverage the themes in these two theoretical constructs to describe the qualitative data.

Despite the survey that we used to assess developers' perception of the security feedback beginning with an informed consent form, one of the developers remarked that they have "observed a worrying trend in the research community of sending unsolicited surveys to open source project contributors" and that they "did not consent to be the subject of this research." The developer suggested, in retrospect, that we should "solicit feedback from the community on whether or not it is interested in making this a part of the code review process" before posting comments that may be regarded as "disruptive" and "spam" which may in turn lead to "accounts blocked."

Developers expected the feedback to be *specific*, *actionable*, and in the *context* of the change. Developers expected the feedback to be "specific and actionable" and that "stating that a particular file scored high on some model is of absolutely no use to us at all." The feedback must include "definitive evidence ... (not ... probability) that there is a bug" and "some indication of where it is" and, if definitive evidence is not available, a description of the ways in which developers "could improve the change" and/or a prescription of "what action should a developer take". The feedback must be specific to the change in that it must "discuss the actual change" and if the change is actually "causing any security issues".

Chromium developers expect the feedback, and by extension, the vulnerability discovery metrics the insights from which informed the feedback, to be *specific*, *actionable*, and in the *context* of the change. The expectations are largely informed by the developers' preference for the feedback to lead to a bug report, one that they can triage and fix.

The observations from the expectations research question highlights a possible disconnect in the definition of actionability between research and practice. As Meneely et al. [84] mention, the perception of actionability differs between theory-driven and goal-driven philosophies, with those of the theory-driven philosophy not regarding a metric as useless simply because it is not actionable because some metrics tend to be relevant to characterize attributes of software beyond specific goals. Developers, being goal-driven, expect the metrics to be actionable in the sense that the metric must lead to a tangible goal (the creation of a bug report describing the vulnerability that the metric found in a piece of software, in our case). Vulnerability discovery metrics quantify *systemic problems* within the product, process, or people involved in engineering software and literature has shown that such systemic problems are associated with historical vulnerabilities. However, Chromium developers' expectation of *specific*, *actionable*, and in the *context* of the change do not seem likely to be fulfilled by vulnerability discovery metrics. The only exception to this observation was the expectation from one of the developers that the metric indicate that "this file you are touching right now has had security problems in the past" and the (known) offender metric [85] fulfills this expectation.

### 6.3.3 RQ 7 - Effectiveness

Question: *How effective are existing vulnerability discovery approaches?*

**Motivation.** In addressing the expectations research question, we observed that developers, being of the goal-driven philosophy, expect their vulnerability discovery approaches to be specific, actionable, and in the context of the change. We further observed that the vulnerability discovery metrics we used are unlikely to fulfill these expectations. In the effectiveness research question, we wanted to broaden the scope of the vulnerability discovery beyond metrics to include other, existing, vulnerability discovery approaches to identify those that are likely to meet developers' expectations.

Shahriar and Zulkernine [130] systematically reviewed the literature to compare and contrast various approaches proposed to mitigate vulnerabilities in software, including an enumeration of the challenges in applying these mitigation approaches. Among the three vulnerability mitigation techniques surveyed, static analysis achieved the same coverage of vulnerabilities as testing and hybrid analysis (i.e. combination of static and dynamic analysis). Despite the high false positive rate [56], the use of static analysis techniques for security [19] has seen a steady growth with several popular open-source projects using at least one static analysis tool to analyze their source code [8]. The concern of static analysis breaking developers' workflows [52] is being alleviated by solutions like Tricorder [126] and Tricium [41] by Google and GitLab DevSecOps [40] seamlessly integrating static analysis into existing developer workflows.

The motivation for the effectiveness research question is to assess the effectiveness of existing vulnerability discovery approaches in assisting developers discover vulnerabilities. Among the existing vulnerability discovery approaches, we choose to target static analysis for the following reasons: (1) static analysis does not require the source code to be executed, freeing our analysis from the need to compile the source code of our study subjects, (2) there are several popular open-source static analyzers that we could use in our analysis,

and (3) with static analysis increasingly becoming a commonplace in most mature software engineering workflows, our analysis could provide insights on its effectiveness in vulnerability discovery.

**Approach.** We address the effectiveness research question in the context of the FFmpeg project. We chose FFmpeg because the dataset of vulnerabilities that have been fixed in the project is sizable yet not prohibitive when identifying vulnerability contributing commits. We begin the analysis by identifying the parent commit of each of the vulnerability contributing commits in FFmpeg. The parent of a vulnerability contributing commit represents the state of the source code before the vulnerability was contributed (by the vulnerability contributing commit). We checkout the FFmpeg source code at each vulnerability contributing commit, and its parent, and subject it to static analysis (See Section 6.2.1 for details about static analysis as used in our work).

As mentioned earlier, a vulnerability contributing commit can contribute to a vulnerability in one of two ways: the vulnerability is contributed when the file is added for the first time or the vulnerability is contributed when the file is modified during its evolution. The dichotomy of vulnerability contributing commits based on the mode of vulnerability contribution is necessary because the analysis approach, as described below, is different based on the mode of vulnerability contribution.

*Vulnerability Contributed During File Modification*

In this case, we compare the static analysis findings associated with the vulnerable file at vulnerability contributing commit and that at its parent. The expectation here is that the change to the vulnerable file must be correlated with an increase in the number of static analysis findings.

The analysis approach is slightly involved because the task of identifying static analysis findings attributable to code modified in the vulnerability contributing commit is nontrivial. We illustrate this with an example shown in Table 6.3. In the example, `foo.c` is an existing file into which a vulnerability was introduced at line 4 by the vulnerability contributing commit, `contributor`. `foo.c` before (i.e. at commit `parent`) the vulnerability was introduced, had a trivial static analysis finding (unused variable) on line 4. Since the vulnerability contributing commit added a new line to `foo.c`, line 4 in `parent` is now line 5 in `contributor`. If we were to subject `foo.c` at `contributor` to static analysis, two findings would be reported, one at line 4 (new in `contributor`) and one at line 5 (unresolved from `parent`). The actual number of static analysis findings attributable to the code modified in `contributor`, however, is just the one associated with line 4. In our analysis, we used the patch associated with the vulnerability contributing commit to map a line after the commit was applied to the line before the commit was applied.

Table 6.3: Two versions, `contributor` and `parent`, of a file, `foo.c`, with `contributor` contributing to a vulnerability used to illustrate the approach to identify static analysis findings associated with code modified in a vulnerability contributing commit

| contributor | | parent | |
|---|---|---|---|
| **Code** | **Line** | **Line** | **Code** |
| `#include <stdio.h>` | 1 | 1 | `#include <stdio.h>` |
| `int main(int argc, char *argv[]) {` | 2 | 2 | `int main(int argc, char *argv[]) {` |
| `    char message[6] = "world";` | 3 | 3 | `    char message[6] = "world";` |
| `    gets(message);` | 4 | - | |
| `    int x;` | 5 | 4 | `    int x;` |
| `    printf("hello %s\n", message);` | 6 | 5 | `    printf("hello %s\n", message);` |
| `    return 0;` | 7 | 6 | `    return 0;` |
| `}` | 8 | 7 | `}` |

An alternative, arguably simpler, approach would have been to only consider the static analysis findings associated with the vulnerable file after the vulnerability contributing commit was applied to it and identify those findings that were associated with the suspect lines. Perl et al. [115] did indeed use this approach in their work. The intention with our choice of analysis approach was to account for the situation where the lines of code being added, deleted, and/or modified in the vulnerability contributing commit causes the introduction of static analysis findings in a different part of the file. In effect, our approach is attempting to improve recall over the alternative (simpler) approach.

*Vulnerability Contributed During File Addition*

In this case, we reason about the distance, in number of lines, between the suspect lines and the static analysis finding lines. The expectation here is that the vulnerable file must have static analysis findings near the lines suspected of contributing to the vulnerability.

As with our analysis approach for vulnerabilities contributed during file modification, we are not only assessing if static analysis findings were associated with the vulnerable lines but the distance of the static analysis findings from the vulnerable lines, achieving a holistic perspective over the alternative approach [115]. Furthermore, the distance affords us the ability to reason about the likelihood of the developer even having seen the vulnerable lines in their peripheral view, if not directly, when perusing the static analysis findings. For instance, if a file with 120 lines of code had a finding at line 20 but the vulnerable line of code was at line 118, the developer is unlikely to have even seen the vulnerable line, which was 98 lines away from the static analysis finding.

The approach to compute the distance between suspect lines and the static analysis finding lines involves an aggregation step because a single vulnerable file can have more than one suspect lines and more than one static analysis finding lines. We compute the distance between a single static analysis finding line and all the suspect lines and aggregate the distances at the static analysis finding level. For instance, consider a file `foo.c` which has two static analysis findings at lines 53 and 64 and three suspect lines at 13, 34, and 82. We first compute the distance between the each static analysis finding and each of the three suspect lines to get 40, 19, and 29 for finding at line 53 and 51, 30, 18 at line 64. We then aggregate the distances computed for each static analysis finding to get the minimum distance between the finding and the suspect line. If the minimum distance is zero, then the static analysis finding is on the same line as one of the suspects.

**Observations.** We begin the presentation of our observations by summarizing the static analysis findings associated with known vulnerable files at the time of vulnerability contribution (i.e. at the vulnerability contributing commit). Shown in Figures 6.2 and 6.3 is the distribution of number of static analysis findings in known vulnerable files at the time of vulnerability contribution and their severity, respectively.

The distribution of the number of static analysis findings (Figure 6.2) is clearly right skewed, implying that several files have few findings while a few files have several findings. We assessed if the skewness of the distribution could be an artifact of the size of the vulnerable file. Shown in Figure 6.4 is a scatter plot depicting the correlation between the size of the vulnerable file (expressed as number of source lines of code) and the number of static analysis findings associated with the file. A trivial inference one can make from Figure 6.4 is that the size of vulnerable file is correlated with the number of static analysis findings associated with the file. We found quantitative evidence to support this inference leading us to conclude that vulnerable files with more source lines of code are statistically (Spearman's p-value $\ll 0.01$) and practically (Spearman's $\rho = 0.7999$ representing a strong positive correlation) more likely to be associated with more static analysis findings. As a consequence of this observation, we

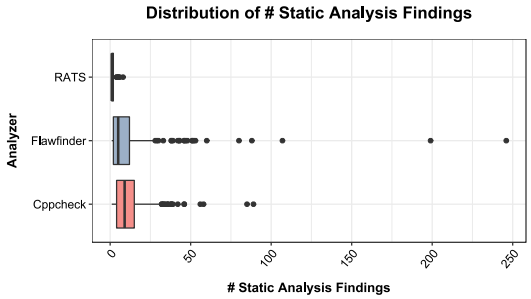



Figure 6.2: Distribution of number of static analysis findings in known vulnerable files at the time of vulnerability contributing commit

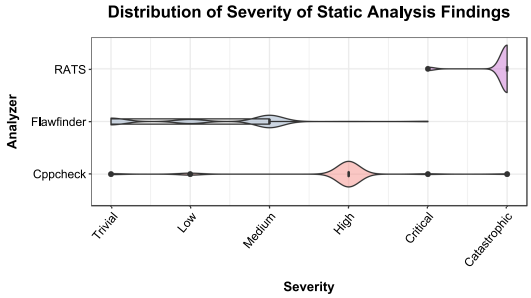



Figure 6.3: Distribution of severity of static analysis findings in known vulnerable files at the time of vulnerability contributing commit

use density (ratio of number to size) of static analysis findings rather than the number.

The distribution of the severity of static analysis findings (Figure 6.3) indicates that the static analysis findings from the different analyzers are very different, with RATS being left skewed and Flawfinder being right skewed. The skewness could be an artifact of the difference in the way the analyzers assign severity to the findings or it could mean that certain analyzers are better at discovering fewer high severity problems while others are better at discovering several low severity problems. If it is the latter, then our decision to use multiple static analyzers to improve soundness [19] of static analysis, as suggested by Li and Cui [65], is supported by the data from FFmpeg.

We now present the observations from applying our approach to addressing the effectiveness research question. Since the analysis approach is different if the vulnerability was contributed when the vulnerability contributing commit modified an existing file and when it added a new file, we present the observations separately as well.

*Vulnerability Contributed During File Modification*

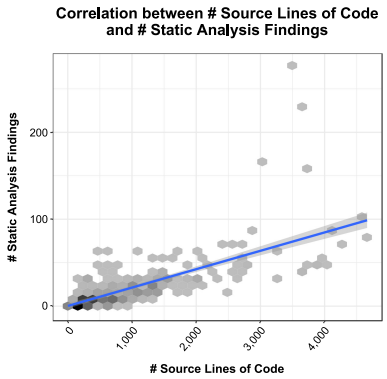



Figure 6.4: Scatter plot depicting the correlation between the size of the vulnerable file (expressed as number of source lines of code) and the number of static analysis findings associated with the file at the time of vulnerability contributing commit

In the case of vulnerabilities contributed during file modification, we have two versions of the vulnerable file: before and after the vulnerability was contributed by the vulnerability contributing commit. The key in assessing the effectiveness of static analysis to highlight risk in these commits is to identify if there is a change in the number of static analysis findings associated with the vulnerable file before and after the commit. In other words, we assess if a vulnerability contributing commit contributes *new* static analysis findings to the vulnerable file.

Shown in Figure 6.5 is the distribution of the severity of static analysis findings associated with vulnerable files at the time of vulnerability contribution during file modification. An interesting insight from Figure 6.5 is that the severity of *unresolved* static analysis findings (i.e. static analysis findings that existed even before the vulnerability contributing commit) is high (mean severity was 3.16), implying that the files were already at risk of being vulnerable.

If static analysis was an effective means of vulnerability discovery, then every vulnerability contributing commit must be associated with at least one *new* static analysis findings. We evaluate this notion by computing the percentage of vulnerability contributing commits which were associated with at least one *new* static analysis finding. Shown in Table 6.4 is the percentage of vulnerable files marked as risky (i.e. have at least one *new* static analysis finding) after a vulnerability was contributed by the vulnerability contributing commit. If developers were to use static analysis to discover vulnerabilities, they are likely to ignore less severe findings (such as unused variables) in favor of the more severe findings (such as null pointer access). We simulate this behavior of developers by stratifying the static analysis findings by severity. The minimum severity in Table 6.4 specifies the severity threshold used when computing the percentage of vulnerable files marked as risky. For instance, if the developers had chosen to consider only those static analysis findings with severity of high or above, then 28.57% of the vulnerable files would have been highlighted as being risky by static analysis.

As shown in Table 6.4, the percentage of vulnerable files that would have been marked

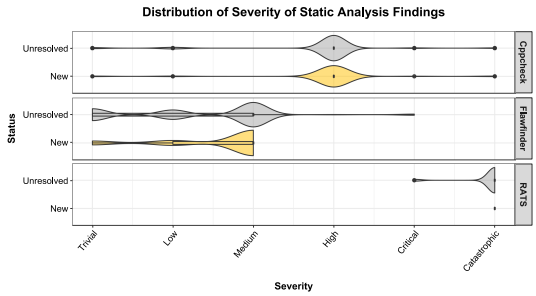



Figure 6.5: Distribution of static analysis findings density associated with vulnerable files at the time of vulnerability contribution during file modification

Table 6.4: Percentage of vulnerable files marked as risky (i.e. have at least one *new* static analysis finding) after a vulnerability was contributed by the vulnerability contributing commit

| Minimum Severity | | | | | |
|---|---|---|---|---|---|
| **Trivial** | **Low** | **Medium** | **High** | **Critical** | **Catastrophic** |
| 36.55% | 35.29% | 34.03% | 28.57% | 4.20% | 3.36% |

as risky by static analysis is low (with 36.55% being the highest value) indicating that a considerable portion of vulnerable files (with 63.45% being the lowest value) would not have gotten the developers' attention for a security review.

Static analysis was found to not highlight risk of vulnerability in 63.45% of vulnerable files in the FFmpeg project at the time of vulnerability contributing commit.

*Vulnerability Contributed During File Addition*

In the case of vulnerabilities contributed during file addition, we have a single version of the vulnerable file. As a result, all static analysis findings associated with the vulnerable file are *new*. As shown in Figure 6.6, the density of static analysis findings in vulnerable files is low (mean density was 0.0319 i.e. on average, there were approximately three static analysis findings for every one hundred source lines of code). While certain static analysis findings (like unused variable) may be trivial to address due to their narrow scope, others may require the developer to have to review additional lines of code to ensure that addressing a static analysis finding does not lead to additional bugs. The low density of static analysis findings indicates that a developer is likely to have to review a considerable number of lines of code when addressing few static analysis findings.

If static analysis was an effective means of vulnerability discovery, then every vulnerability contributing commit must be associated with static analysis findings that are *near* the lines that were suspected of contributing to the vulnerability. Shown in Figure 6.7 is

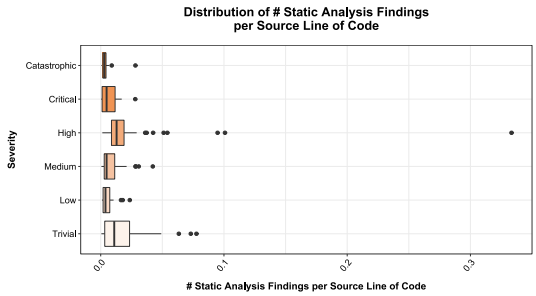



Figure 6.6: Distribution of density of static analysis findings associated with vulnerable files at the time of vulnerability contribution during file addition

the distribution of minimum distance between a static analysis finding and suspect lines of code in vulnerable files. The variance in minimum distance is quite large. Furthermore, the right skewness of the distributions indicates that a considerable number of static analysis findings are near the suspect lines. However, the mean and median of the minimum distance between a static analysis finding and suspect lines of code shown in Table 6.5 indicate that the minimum distance is not quite near enough. If developers were to review every line in the n (where n is the mean of minimum distance in Table 6.5) lines before and after the static analysis finding, then the they would need approximately one and a half hour[14] to address each static analysis finding with trivial severity.

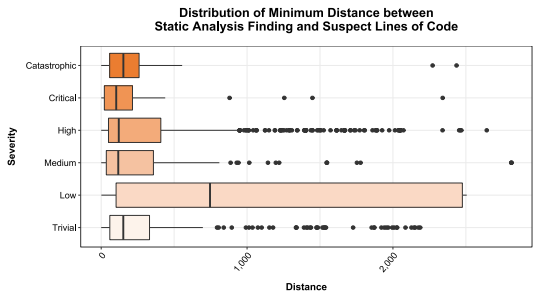



Figure 6.7: Distribution of minimum distance between a static analysis finding and suspect lines of code in vulnerable files at the time of vulnerability contribution during file addition

| |
|---|
| Static analysis findings were found to not be near lines suspected of contributing a vulnerability in the FFmpeg project at the time of vulnerability contributing commit. |

[14] Using 200 lines of code per hour as the rate of review [53]

Table 6.5: Mean and median of the minimum distance between a static analysis finding and suspect lines of code in vulnerable files at the time of vulnerability contribution during file addition

| Severity | Minimum Distance | |
|---|---|---|
| | Mean | Median |
| Trivial | 395 | 152 |
| Low | 1,214 | 746 |
| Medium | 267 | 117 |
| High | 358 | 120 |
| Critical | 235 | 103 |
| Catastrophic | 292 | 152 |

### 6.3.4 RQ 8 - Utility

Question: *Is there a utility for vulnerability discovery metrics?*

**Motivation.** In addressing the effectiveness research question, we observed that static analysis, a commonly-used approach to improve the overall quality of software [8], (1) would have missed to highlight risk of a vulnerability in 63.45% of vulnerable files when the files were being modified and (2) the findings reported were not near the lines suspected of being vulnerable in vulnerable files when the files were being added. In the utility research question, we wanted to assess the utility of vulnerability discovery metrics in highlighting risk in vulnerability contributing commits.

Although there are a plethora of approaches to discover vulnerabilities in software [39], the academic investigation of the effectiveness of these approaches has largely been siloed. Over the years, there have only been a few studies that have attempted to assess the complementary aspects of leveraging multiple vulnerability discovery approaches to assist developers in improving security of software. As recently as 2018, researchers [139] have highlighted the need to investigate the ways in which one vulnerability discovery approach can complement another in improving the effectiveness of vulnerability discovery. In our review, we found a study by Gegick et al. [37] to be the only one to have investigated the complementary aspects of vulnerability discovery metrics and static analysis. In their study of pre- and post-release failures, Gegick et al. [37] found that metrics (churn and source lines of code) alone were ineffective at accurately predicting attack-prone components but a combination of metrics and automated static analysis alert density predicted 100% of attack-prone components with an 8% false positive rate.

The motivation for the utility research question is to assess the utility of vulnerability discovery metrics in highlighting risk from vulnerability contributing commits. We use the term *utility* here as a catchall for the various ways in which the metrics can be reasoned about as being useful, the chief among which is to highlight risk from vulnerability contributing commits.

**Approach.** As with the effectiveness research question, we address the utility research question in the context of the FFmpeg project. We begin by assessing if the vulnerability

discovery metrics would have even highlighted vulnerability contributing commits in FFmpeg as being risky.

We aggregate the risk associated with individual vulnerability discovery metric value at the appropriate level of granularity to enable reasoning about risk at the higher level of granularity. For instance, we aggregate risk associated with all change-level metrics to express change risk. We use median of individual risk to achieve this aggregation. We then aggregate the risk associated with the change, commit, file, and developer to express the risk associated with the vulnerability contributing commit. We use the maximum of change, commit, developer, and file risks to achieve this aggregation to ensure granular risk is propagated to the vulnerability contributing commit. The aggregation approach is pictorially represented as a top-down tree shown in Figure 6.8. The colored circles with numbers in Figure 6.8 are sample risk levels that serve as an example to aid the comprehension of our aggregation approach.

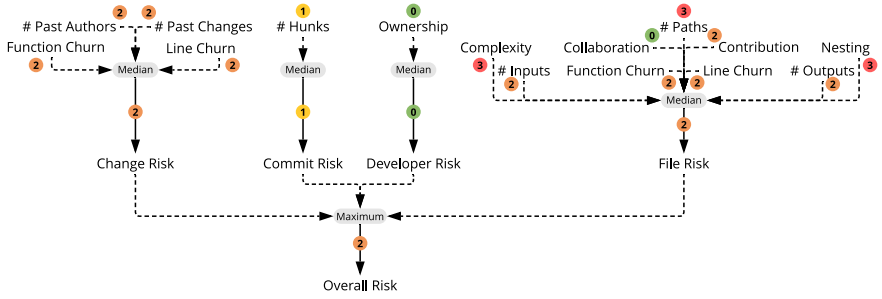



Figure 6.8: Pictorial representation of the approach used to aggregate risk associated with metrics at individual levels of granularity (change, commit, developer, and file) and to aggregate the change, commit, developer, and file risk to express the overall risk. The colored circles with numbers are sample risk levels that serve as an example.

As with the effectiveness research question, we identify, and use, the parent each of the vulnerability contributing commits to represent the state of the source code before the vulnerability was contributed. We collected the 18 metrics described in Section A.2 of Appendix A from FFmpeg before and after each vulnerability contributing commit. While we collected all 18 metrics, we did not consider three commit-level metrics—message tokens, patch tokens, and patch keyword frequency—when addressing the utility research question due to the inherent lack of generalizability of these metrics. We also do not consider the source lines of code metric as a vulnerability discovery metric because we used it to normalize other file-level metrics when computing the thresholds. In effect, we consider the following metrics: function churn, line churn, # past authors, and # past changes at the change level, # hunks at the commit level, ownership at the developer level, and collaboration, complexity, contribution, function churn, line churn, nesting, offender, # inputs, # output, and # paths at the file level. Although we considered the offender metric, We did consider the offender metric at the file level but since it is a boolean-valued metric, it has no threshold except that its value itself indicates if the change is risky or not.

We used the metrics collected from FFmpeg before a vulnerability contributing commit to compute the metrics' thresholds and used the thresholds to associate appropriate risk

levels to the metrics collected from FFmpeg after the vulnerability contributing commit. The specifics of computing metric thresholds and using the thresholds to classify risk is described in Section 6.2.2. As mentioned earlier, we use two different versions of FFmpeg because we wanted to simulate the likely scenario in which the thresholds computed in the past (before a vulnerability contributing commit, in our case) are used to assess risk of a future change (the vulnerability contributing commit, in our case).

Similar to the effectiveness research question, the analysis approach here is dependent on the mode of vulnerability contribution: vulnerability contributed during file modification or vulnerability contributed during file addition. The dichotomy of vulnerability contributing commits based on the mode of vulnerability contribution is necessary because the set of vulnerability discovery metrics available are different based on the mode of vulnerability contribution. The reason for this difference in vulnerability discovery metric availability is because some metrics (for instance, contribution centrality and collaboration centrality proposed by Meneely and Williams [81]) are based on the history of a file and such metrics are not defined when analyzing a vulnerability contributed during file addition.

**Observations.** As described in the approach to addressing the utility research question, we analyzed vulnerability contributing commits that contributed a vulnerability during file modification separately from those that contributed a vulnerability during file addition. As a result, we present our observations separately as well.

*Vulnerability Contributed During File Modification*

When analyzing vulnerability contributing commits that contributed a vulnerability during file modification, all 16 metrics listed in the approach are defined.

At the outset, we observed that the vulnerable file in about one third (29.47%) of the vulnerability contributing commits was a known offender. In their study, Meneely et al. [85] observed that the 26.6% of vulnerability contributing commits in the Apache HTTP Server project had known offenders. The similarity in proportion of vulnerability contributing commits that modified a known offender observed in both studies lends further credence to the (known) offender metric. The observation implies that even a simple metric like (known) offender has the potential to assist developers in engineering secure software by highlighting risky changes and addresses the "... metric that would help me would be "this file you are touching right now has had security problems in the past"." expectation that one of the Chromium developers expressed.

| A vulnerable file in about one third (29.47%) of the vulnerability contributing commits that contributed a vulnerability during file modification was a known offender. |
|---|

Shown in Figure 6.9 is percentage of vulnerability contributing commits which contributed a vulnerability during file modification highlighted as risky at non-trivial metric risk levels. The percentage values shown in Figure 6.9 are cumulative; for instance, the 34.3% associated with high change risk includes those changes that had high and critical risk.

As can be inferred from the percentages in Figure 6.9, a considerable portion of the vulnerability contributing commits had at least a medium change, commit, and file risk. Developer risk, on the other hand, is rather interesting with only 3.4% of the vulnerability contributing commits being risky at all risk levels. A factor that is likely to have contributed to such a low proportion is the skewness in the distribution of ownership (the only developer-level metric in our study). Shown in Figure 6.10 is the distribution of ownership in the FFmpeg project at the time of the most recent vulnerability contributing commit (`c8c81ac502`). The distribution is clearly right skewed to a severe degree with the skew

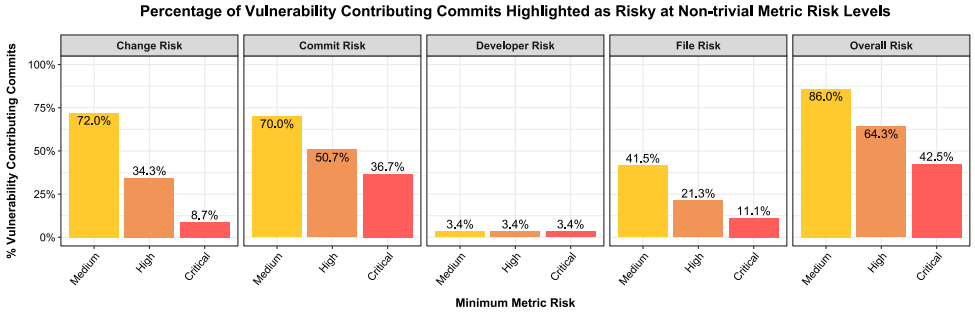



Figure 6.9: Percentage of vulnerability contributing commits which contributed a vulnerability during file modification highlighted as risky at non-trivial metric risk levels

quantitatively estimated to be 34.95 [51]. The skewness indicates that a large number of developers have contributed few changes to the FFmpeg project. In fact, 90% of the commits to FFmpeg at `c8c81ac502` were contributed to by 5.93% of the developers. A direct consequence of the skew in distribution of ownership is that the thresholds computed for the metric tends to not capture the lack of variance in the metric values.

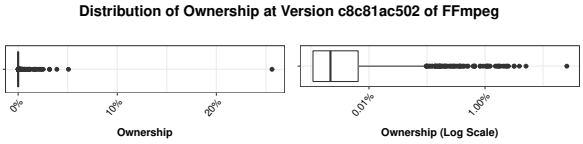



Figure 6.10: Distribution of ownership in the FFmpeg project at the time of the most recent vulnerability contributing commit (`c8c81ac502`)

In the remainder of the discussion here, we only reason about risk at the critical level. By reasoning at critical level of risk, we constrain ourselves to the top 10% of changes, commits, and files. For instance, the threshold value at the 90% quantile for the # hunks commit-level metric in the FFmpeg project before the most recent vulnerability contributing commit (`c8c81ac502`) is 8. We know (See Section 6.2.2 for specifics on the approach used to compute thresholds) that at most 10% of the commits can have a # hunks value higher than 8.

At the individual level of granularity, commit risk seems to be most effective at highlighting risk in vulnerability contributing commits with 36.7% being assessed as having critical risk. The commit risk, having been aggregated from a single commit-level metric (# hunks), indicates that vulnerability contributing commits tend to have very high value for the # hunks metric which in turn indicates that the commit is highly fragmented [115]. Change risk is not as effective with only 8.7% of vulnerability contributing commits being assessed as having critical risk. However, change risk is aggregated from the most granular metrics asso-

ciated with a commit (i.e. function churn, line churn, # past authors, and # past changes) meaning the risk highlighted by the change risk is likely to be immediately understandable by a developer contributing the change. File risk, as with change risk, does not seem to be as effective highlighting only 11.1% of the vulnerability contributing commits as being risky. However, file risk has been aggregated from nine individual metrics collected at the file level and the aggregation strategy (median, in our case) may have had an impact on the effective file risk. The simplicity of the risk assessment approach means that practitioners can change the aggregation strategy from median to, say, weighted average to selectively prioritize certain metrics over others.

The overall risk, aggregated from change, commit, developer, and file risks, is effective at highlighting 42.5% of the vulnerability contributing commits as risky.

| |
|---|
| Overall, risk from metrics was effective at highlighting 42.5% of the vulnerability contributing commits as being risky at the critical level. However, at individual levels of granularity, commit risk was most effective, highlighting 36.7% of the vulnerability contributing commits as risky followed by file risk with 11.1%, change risk with 8.7%, and developer risk with 3.4% |

In the effectiveness research question, we observed that static analysis highlighted 36.55% of the vulnerability contributing commits as risky (See Table 6.4). In terms of absolute values, the overall metric risk highlighted 42.5% of the vulnerability contributing commits as risky outperforming static analysis. While the comparison is not fair because of the levels of granularity at which the analysis was performed for static analysis. In assessing the effectiveness of static analysis, we classified a vulnerability contributing commit as risky if the vulnerable file had at least one new static analysis finding implying that the new static analysis finding was contributed by the change. The overall metric risk is aggregated from risk at individual levels of granularity, two of which are not directly associated with the change itself. However, if we considered only the change risk and commit risk, commit risk still outperforms static analysis (36.7% versus 36.55%). Furthermore, the 36.55% observed for static analysis was a liberal assessment because we considered any new finding as contributing risk, irrespective of its severity, but with metric risk, we considered only risk at the critical level.

| |
|---|
| Overall metric risk at the critical level outperformed static analysis (even with no consideration of severity of findings) in highlighting vulnerability contributing commits as risky. |

We further analyzed the utility of metrics in relation to static analysis by assessing the overall metric risk of those vulnerability contributing commits that static analysis missed (i.e. the 63.45% of the vulnerability contributing commits that had zero new static analysis findings in the vulnerable file after the vulnerability contributing commit). We observed that 32.12% of these missed vulnerability contributing commits had critical overall metric risk. If the overall metric risk was used as a secondary risk assessment mechanism (with static analysis being the primary), 68.67% = 36.55% (static analysis) + 32.12% (metric) of the vulnerability contributing commits would have found to be risky.

| |
|---|
| Overall metric risk was effective at highlighting 32.12% of the vulnerability contributing commits missed by static analysis as risky. In effect, a combination of static analysis and metric risk would have highlighted 69% of the vulnerability contributing commits as risky. |

The increase in percentage of vulnerability contributing commits that would have been highlighted as risky with the inclusion of metric risk presents a promising avenue for using

static analysis as the primary line of risk assessment with metric being the supplemental, secondary line of risk assessment.

*Vulnerability Contributed During File Addition*

When analyzing vulnerability contributing commits that contributed a vulnerability during file addition, 9 of the 16 metrics listed in the approach are defined. These 9 metrics are function churn and line churn at the change level, # hunks at the commit level, ownership at the developer level, and complexity, nesting, # inputs, # outputs, and # paths at the file level. The loss of 7 out of 16 metrics (approximately 44%) is a considerable limitation and goes to show that metrics that are dependent on the history of a file have no utility in highlighting risk in commits that contribute vulnerabilities when adding a file.

Shown in Figure 6.11 is percentage of vulnerability contributing commits which contributed a vulnerability during file addition highlighted as risky at non-trivial metric risk levels. The percentage values shown in Figure 6.11 are cumulative; for instance, the 98.1% associated with high change risk includes those changes that had high and critical risk.

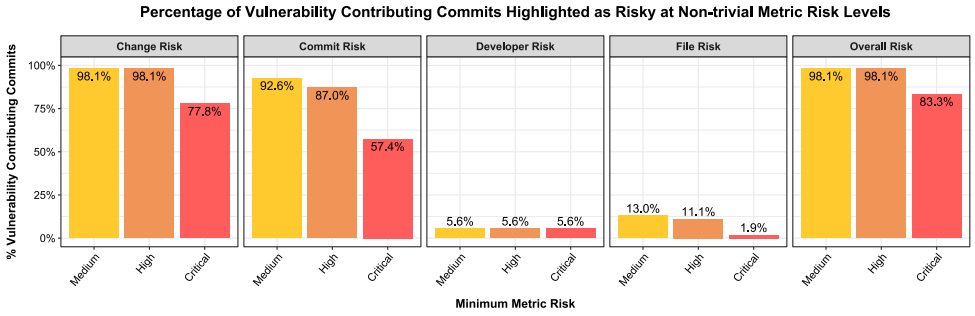



Figure 6.11: Percentage of vulnerability contributing commits which contributed a vulnerability during file addition highlighted as risky at non-trivial metric risk levels

As can be inferred from the distribution in Figure 6.11, more than half of the vulnerability contributing commits had critical risk at the change and commit levels. Developer risk only highlighted a 5.6% of the vulnerability contributing commits as risky. Although the value is higher than that observed when analyzing vulnerability contributing commits that contributed a vulnerability during file modification (See Figure 6.9), it is not high enough to be effective. However, we did analyze if the ownership of developers contributing vulnerabilities during file addition is lower than that of developers contributing vulnerabilities during file modification. We used comparative box plot, shown in Figure 6.12, to qualitatively assess if there is an association between ownership and the mode of vulnerability contribution. The comparative box plot shows a considerable overlap between the distributions of ownership of developers contributing vulnerabilities during file addition and that of developers contributing vulnerabilities during file modification. The quantitative assessment supported the qualitative inference that can be drawn from Figure 6.12 revealing no statistical evidence (Mann-Whitney-Wilcoxon p-value $> 0.05$) to conclude that the two distributions are indeed different in the FFmpeg project.

The file risk hardly highlighted any vulnerability contributing commits as risky which

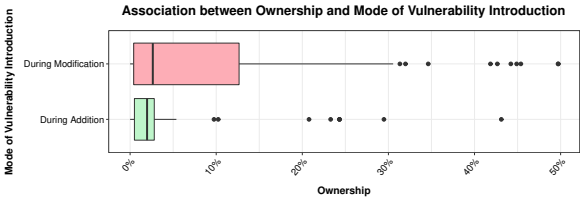



Figure 6.12: Comparing the distribution of developer ownership in vulnerability contributing commits in which a vulnerability was contributed during file modification and that in vulnerability contributing commits in which a vulnerability was contributed during file addition

could mean that the file-level metrics defined on a new file may not be as utilitarian in highlighting risky commits than metrics defined on an existing file.

The overall risk, aggregated from change, commit, developer, and file risks, is effective at highlighting 83.3% of the vulnerability contributing commits as risky. While the high percentage may seem impressive, the overall risk is likely being biased by change risk and commit risk, which by themselves seem effective. The reason for change risk and commit risk to be highlighting so many of the vulnerability contributing commits as risky is because these commits are adding the vulnerable file which, by the nature of addition, will have very high values for change-level and commit-level metrics as compared to a file that is being modified. Nevertheless, if the vulnerability contributing commit is required to be subject to a code review prior to its acceptance into the source code repository, the file being added must be assumed to be risky simply because of the size of the change.

Overall, risk from metrics was effective at highlighting 83.3% of the vulnerability contributing commits as risky. However, the effectiveness may be affected by the fact that a commit adding a file is likely to have very high values for change-level and commit-levels metrics relative to a commit modifying a file.

## 6.4 Summary

In this study, we considered the notion that "un-actionable (metric) is not useless" [84] and assessed the utility of metrics, treated as agents of feedback rather than mere features in a black box prediction model, in assisting developers discover vulnerabilities. We used values, (unsupervised) thresholds, and interpretation of metrics collected from the Chromium project to generate natural language feedback on security. We provided this feedback to Chromium developers and assessed their perception using a survey. In conducting the study, we observed that the Chromium developers reported the insights from vulnerability discovery metrics, despite being read, understood, and discussed, to be not surprising, at best, and not useful, at worst. In a follow up conversation with the developers, we discovered that their expectations were largely informed by the preference for the feedback to lead to a bug report, one that they can triage and fix, rather than understanding the potential for deeper systemic problems. We assessed if existing vulnerability discovery approach—static

analysis—which was likely to meet the developers' expectation was utilitarian in discovering vulnerabilities. We found that static analysis missed to highlight a risk in nearly two-thirds of the vulnerability contributing commits in FFmpeg. However, metrics, when used independently, highlighted risk in a little over two-fifths of the vulnerability contributing commits and, when used in concert with static analysis, highlighted risk in little over to two-thirds of the vulnerability contributing commits. We discovered that the metrics did indeed contribute an aspect to vulnerability discovery that static analysis by itself could not.

# Chapter 7

# Summary

In our pursuit of fine-grained metrics to discover vulnerabilities in software, we proposed two empirically-validated metrics—proximity and risky walk—based on the attack surface model of a software system. In empirically validating proximity and risky walk metrics, we demonstrated that the prediction model built with these metrics as explanatory variables outperformed comparable models from prior literature. The absolute performance of the model, however, left a lot to be desired. In the applying a conventional approach to using the performance of a vulnerability prediction model to infer the utility of metrics, we are ignoring the metrics' ability to tell a story, as Fenton and Neil suggested in their software metrics roadmap almost twenty years ago [32]. Furthermore, as Meneely et al. [84] mention, "un-actionable (metric) is not useless" to a person (researcher or practitioner) of a theory-driven persuasion. In effect, we were falling short of asking ourselves *what is the metric telling us?* and *what can we ask developers to do?*. The introspection on conventional practice of rejecting metrics based solely on their inability to predict vulnerabilities in the context of a black box model proved to be the catalyst behind the vision of the work presented in this dissertation.

The vision of our research presented in this dissertation was *to assist software engineers in building secure software by providing a technique that generates scientific, interpretable, and actionable feedback on security as the software evolves.* We presented our observations from two research studies that we conducted toward achieving this vision. We began by systematically reviewing the literature to enumerate the vulnerability discovery metrics that have been proposed and/or evaluated in the literature. We identified a plethora of metrics but the validity of the metrics has primarily been evaluated in the context open-source subjects written in few programming languages with limited replication by authors other than the ones proposing the metrics. We also observed that the validation of the metrics tended to be biased toward assessing association, discrimination, and prediction, with limited attention paid to assessing causality or actionability. Furthermore, the metrics were seldom validated using real developers. We implemented a subset of the metrics identified during the review and specifically assessed the generalizability of metrics across projects, application domains, and programming languages. We used an unsupervised approach from literature to compute metric thresholds and assessed the ability of the thresholds to classify risk from historical vulnerabilities in an open-source project. In keeping with our vision to provide developers

with security feedback, we leveraged the metric values, thresholds, and interpretation to provide developers with data-driven feedback on security as they contributed changes to a large open-source project. In seeing that our feedback was being perceived as spam, we initiated an open dialogue with the developer community to ascertain their expectations from such feedback. In response to the comments from the developer community, we assessed the effectiveness of an existing vulnerability discovery approach—static analysis—that was likely to fulfil developers' expectations. However, we found that the existing vulnerability discovery approach highlight about one-third of commits that likely contributed a vulnerability as being risky. We assessed the utility of the metrics to highlight risk in the same set of vulnerability contributing commits and found metrics to outperform existing vulnerability discovery approach. Furthermore, we found that static analysis and metrics in concert was able to highlight risk in a considerable portion of vulnerability contributing commits.

In summary, the immediate implications of our work are as follows:

- We highlight, through a systematic literature review of vulnerability discover metrics in Chapter 5, certain shortcomings of the validation approach used to assess the empirical validity of the metrics, in general, and decision-informing ability of the metrics, in particular. The bias toward using open-source subjects of study and relying on association, discrimination, and prediction to assess metrics' decision-informing ability is likely to mask the inherent potential of the metrics: to highlight systemic problems in the product, process, and/or people.

- We present our observations from using the insights from vulnerability discovery metrics to provide feedback to Chromium developers as they contributed changes to the project. We observed that the developers reported that the feedback, despite being read, understood, and discussed, was not surprising, at best, and not useful, at worst. In a subsequent dialogue with the Chromium developers, we observed that developers expected the feedback, and by extension, the vulnerability discovery metrics the insights from which informed the feedback, to be *specific*, *actionable*, and in the *context* of the change. The expectations were largely informed by the developers' preference for the feedback to lead to a bug report, one that they can triage and fix.

- We provide empirical evidence to show that an existing vulnerability discovery approach—static analysis—that satisfied the developers' expectations missed to highlight risk in nearly two-thirds of the vulnerability contributing commits in the FFmpeg project.

- We provide empirical evidence to show that the metrics, when used independently, were effective at highlighting risk in little less than half of the vulnerability contributing commits in FFmpeg. However, when static analysis and metrics were used in concert, little over two-thirds of the vulnerability contributing commits were highlighted as being risky.

- We provide evidence to show that metrics indeed have an utility in highlighting risk in vulnerability contributing commits despite being perceived as not being actionable by developers.

- We present Samaritan metrics platform (described in detail in Appendix A) which comprises of the eighteen vulnerability discovery metrics implemented as containerized microservices to aid replication by researchers and usage by practitioners and the website (`https://samaritan.github.io/`) which is the venue of dissemination of our research to the broader software development community.

## 7.1 Future Work

In the conducting the research studies in pursuit of achieving our research vision, we have gained newfound insights into the expectations of developers from security feedback derived from vulnerability discovery metrics. We, unsurprisingly, observed that developers, being goal-driven, expected the feedback to lead to a concrete bug report that they could act on. Unfortunately, however, the value of metrics is in highlighting deeper systemic problems in the product, process, or people so that developers or managers can foreshadow risk. We expect our to spur interest to further aid the adoption of metrics in practice. We highlight some of the ways in which we envision our work to continue below.

- We conjecture that lack of adoption is likely due to the perceived ineffectiveness of the metrics in *predicting* vulnerabilities. We are actively pursuing a line of research in which we intend to provide the security feedback to stakeholders other than developers and assess their receptiveness of the feedback. Managers of engineering teams are likely a relevant stakeholders to provide the security feedback to since their role involves decision making. At the other end of the software engineering workflow spectrum, we could provide the feedback to participants in bug bounty programs to validate if focusing their attention on risky files can help their hunt for bugs. We suspect that since the goal in hunting for bugs is to maximize profit (i.e. the bounty), the participants may be more receptive of the feedback that could help support their goal.

- In attempting to accomplish our research vision, we have demonstrated the utility of metrics in highlighting risky changes, commits, and/or files. While our goal is to use this knowledge to assist developers improve the security of software, one can envision our approach being used by malicious actors to identify risky changes, commits, and/or files in open-source projects to exploit any latent vulnerabilities. However, we are hopeful that, as attackers begin using metrics to discover and exploit vulnerabilities, developers will take heed and adopt the metrics into their workflows thus rendering the threat from attackers moot.

In addition to the aforementioned streams of research, we also envision, as a pure engineering exercise, to continue building the Samaritan metrics platform to implement additional vulnerability discovery metrics. The eventual vision for the platform is to have it be offered as a service for projects to leverage to identify systemic problems (if any) using metrics.

# Bibliography


[1] National Vulnerability Database - Home. https://nvd.nist.gov/. Accessed: 2017-10-25.

[2] Aesop. The Shepherd Boy and the Wolf - Aesop Fables. http://read.gov/aesop/043.html, 1919. Accessed: 2019-12-16.

[3] Karim Ali and Ondřej Lhoták. Application-Only Call Graph Construction. In James Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, pages 688–712, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-31057-7. doi: 10.1007/978-3-642-31057-7_30. URL http://dx.doi.org/10.1007/978-3-642-31057-7_30.

[4] C.O. Allen, A.A. Chung, B. Truong, and K.K. Yee. Method, System, and Computer Program Product for Providing Real-time Developer Feedback in an Integrated Development Environment, Jul 2009. URL https://www.google.com/patents/US7562344. US Patent 7562344.

[5] T L Alves, C Ypma, and J Visser. Deriving Metric Thresholds from Benchmark Data. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010. doi: 10.1109/ICSM.2010.5609747.

[6] Carl Auerbach and Louise B. Silverstein. *Qualitative Data: An Introduction to Coding and Analysis*. New York University Press, New York, UNITED STATES, 2003. ISBN 9780814707807.

[7] A. Bacchelli and C. Bird. Expectations, Outcomes, and Challenges of Modern Code Review. In *2013 35th International Conference on Software Engineering (ICSE)*, ICSE '13, pages 712–721, May 2013. doi: 10.1109/ICSE.2013.6606617.

[8] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481, March 2016. doi: 10.1109/SANER.2016.105.

[9] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Operating System Enhancements to Prevent the Misuse of System Calls. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, page 174–183, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132034. doi: 10.1145/352600.352624. URL https://doi.org/10.1145/352600.352624.

[10] Ivan Beschastnikh, Mircea F Lungu, and Yanyan Zhuang. Accelerating Software Engineering Research Adoption with Analysis Bots. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, ICSE-NIER '17, pages 35–38, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-2675-7. doi: 10.1109/ICSE-NIER.2017.17. URL `https://doi.org/10.1109/ICSE-NIER.2017.17`.

[11] R Böhme. Vulnerability Markets: What is the economic value of a zero-day exploit? *Chaos Communication Congress*, pages 27–30, Dec 2005. URL `https://events.ccc.de/congress/2005/fahrplan/attachments/542-Boehme2005_22C3_VulnerabilityMarkets.pdf`.

[12] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. SZZ Unleashed: An Open Implementation of the SZZ Algorithm. *CoRR*, abs/1903.01742, 2019. URL `http://arxiv.org/abs/1903.01742`.

[13] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the Characteristics of Vulnerable Code Changes: An Empirical Study. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 257–268, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635880. URL `http://doi.acm.org/10.1145/2635868.2635880`.

[14] Alexandre Boucher and Mourad Badri. Software metrics thresholds calculation techniques to predict fault-proneness: An empirical comparison. *Information and Software Technology*, 96:38–67, 2018. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2017.11.005. URL `http://www.sciencedirect.com/science/article/pii/S0950584916303135`.

[15] Ulrik Brandes and Thomas Erlebach. *Network Analysis: Methodological Foundations*, volume 3418. Springer-Verlag New York, Inc., 2005. ISBN 978-3-540-31955-9. doi: 10.1007/b106453.

[16] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571 – 583, 2007. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2006.07.009. URL `http://www.sciencedirect.com/science/article/pii/S016412120600197X`. Software Performance.

[17] H. Cavusoglu, H. Cavusoglu, and S. Raghunathan. Efficiency of Vulnerability Disclosure Mechanisms to Disseminate Vulnerability Knowledge. *IEEE Transactions on Software Engineering*, 33(3):171–185, March 2007. ISSN 2326-3881. doi: 10.1109/TSE.2007.26.

[18] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[19] B. Chess and G. McGraw. Static Analysis for Security. *IEEE Security Privacy*, 2(6): 76–79, Nov 2004. ISSN 1558-4046. doi: 10.1109/MSP.2004.111.

[20] Istehad Chowdhury and Mohammad Zulkernine. Can Complexity, Coupling, and Cohesion Metrics Be Used as Early Indicators of Vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1963–1969, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605586397. doi: 10.1145/1774088.1774504. URL `https://doi.org/10.1145/1774088.1774504`.

[21] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57 (3):294–313, 2011. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2010.06.003. URL `http://www.sciencedirect.com/science/article/pii/S1383762110000615`.

[22] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294 – 313, 2011. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2010.06.003. URL `http://www.sciencedirect.com/science/article/pii/S1383762110000615`. Special Issue on Security and Dependability Assurance of Software Architectures.

[23] Chromium. Code Reviews - Chromium. `https://chromium.googlesource.com/chromium/src/+/master/docs/code_reviews.md`, 2019. Accessed: 2019-12-18.

[24] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Taylor & Francis, 2013. ISBN 9781134742707.

[25] Carl Counsell. Formulating Questions and Locating Primary Studies for Inclusion in Systematic Reviews. *Annals of Internal Medicine*, 127(5):380–387, 1997. doi: 10.7326/0003-4819-127-5-199709010-00008. URL `http://dx.doi.org/10.7326/0003-4819-127-5-199709010-00008`.

[26] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129, 2000. doi: 10.1109/DISCEX.2000.821514.

[27] Cppcheck Team. cppcheck. `http://cppcheck.net/manual.pdf`, Sep 2019. Version: 1.89.

[28] J. Czerwonka, N. Nagappan, W. Schulte, and B. Murphy. CODEMINE: Building a Software Development Data Analytics Platform at Microsoft. *IEEE Software*, 30(4): 64–71, July 2013. ISSN 0740-7459. doi: 10.1109/MS.2013.68.

[29] M. Doyle and J. Walden. An Empirical Study of the Evolution of PHP Web Application Security. In *2011 Third International Workshop on Security Measurements and Metrics*, pages 11–20, Sept 2011. doi: 10.1109/Metrisec.2011.18.

[30] D. Evans and D. Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, Jan 2002. ISSN 1937-4194. doi: 10.1109/52.976940.

[31] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 752–761, May 2013. doi: 10.1109/ICSE.2013.6606621.

[32] Norman E. Fenton and Martin Neil. Software Metrics: Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 357–370, New York, NY, USA, 2000. ACM. ISBN 1-58113-253-0. doi: 10.1145/336512.336588. URL http://doi.acm.org/10.1145/336512.336588.

[33] Ellen Fineout-Overholt and Linda Johnston. Teaching EBP: Asking Searchable, Answerable Clinical Questions. *Worldviews on Evidence-Based Nursing*, 2(3):157–160, Aug 2005. doi: 10.1111/j.1741-6787.2005.00032.x. URL https://sigmapubs.onlinelibrary.wiley.com/doi/abs/10.1111/j.1741-6787.2005.00032.x.

[34] Sylvie L Foss and Gail C Murphy. Do Developers Respond to Code Stability Warnings? In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 162–170, Riverton, NJ, USA, 2015. IBM Corp. URL http://dl.acm.org/citation.cfm?id=2886444.2886469.

[35] Bill Gates. Memo from Bill Gates - News Center. https://news.microsoft.com/2012/01/11/memo-from-bill-gates/, Sept 2002. Accessed: 2017-10-09.

[36] M Gegick, P Rotella, and L Williams. Predicting Attack-prone Components. In *2009 International Conference on Software Testing Verification and Validation*, pages 181–190, apr 2009. doi: 10.1109/ICST.2009.36.

[37] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. Prioritizing Software Security Fortification Throughcode-Level Metrics. In *Proceedings of the 4th ACM Workshop on Quality of Protection*, QoP '08, page 31–38, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605583211. doi: 10.1145/1456362.1456370. URL https://doi.org/10.1145/1456362.1456370.

[38] Michael Gegick, Pete Rotella, and Laurie Williams. Toward Non-Security Failures as a Predictor of Security Faults and Failures. In *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, ESSoS '09, page 135–149, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642001987. doi: 10.1007/978-3-642-00199-4_12. URL https://doi.org/10.1007/978-3-642-00199-4_12.

[39] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Comput. Surv.*, 50(4):56:1—-56:36, aug 2017. ISSN 0360-0300. doi: 10.1145/3092566. URL http://doi.acm.org/10.1145/3092566.

[40] GitLab. Static Application Security Testing (SAST) | GitLab. https://docs.gitlab.com/ee/user/application_security/sast/, 2020. Accessed: 2020-01-01.

[41] Google. Tricium - Google. https://chromium.googlesource.com/infra/infra/+/master/go/src/infra/tricium, 2020. Accessed: 2020-01-01.

[42] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call Graph Construction in Object-Oriented Languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, page 108–124, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897919084. doi: 10.1145/263698.264352. URL https://doi.org/10.1145/263698.264352.

[43] Mary W. Hall and Ken Kennedy. Efficient Call Graph Analysis. *ACM Lett. Program. Lang. Syst.*, 1(3):227–242, Sep 1992. ISSN 1057-4514. doi: 10.1145/151640.151643. URL `https://doi.org/10.1145/151640.151643`.

[44] Sara Hooshangi, Richard Weiss, and Justin Cappos. Can the Security Mindset Make Students Better Testers? In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 404–409, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2966-8. doi: 10.1145/2676723.2677268. URL `http://doi.acm.org/10.1145/2676723.2677268`.

[45] Michael Howard. Fending Off Future Attacks by Reducing Attack Surface. *MSDN Magazine*, Feb 2003. URL `https://msdn.microsoft.com/en-us/library/ms972812.aspx`.

[46] Michael Howard and Steve Lipner. *The Security Development Lifecycle (SDL): A Process for Developing Demonstrably More Secure Software*. Microsoft Press, 2006. ISBN 0735622140.

[47] Michael Howard, Jon Pincus, and Jeannette M. Wing. *Measuring Relative Attack Surfaces*. Springer, Boston, MA, 2005. ISBN 978-0-387-24006-0. doi: 10.1007/0-387-24006-3_8. URL `http://dx.doi.org/10.1007/0-387-24006-3_8`.

[48] IEEE Computer Society. IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, jan 2010. doi: 10.1109/IEEESTD.2010.5399061.

[49] IEEE Computer Society. Systems and Software Engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, dec 2010. doi: 10.1109/IEEESTD.2010.5733835.

[50] Graylin Jay, Joanne E. Hale, Randy K. Smith, David Hale, Nicholas A. Kraft, and Charles Ward. Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. *Journal of Software Engineering and Applications*, 02(03): 137–143, 2009. doi: 10.4236/jsea.2009.23020. URL `https://doi.org/10.4236/jsea.2009.23020`.

[51] D. N. Joanes and C. A. Gill. Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 47(1):183–189, 1998. doi: 10.1111/1467-9884.00122. URL `https://rss.onlinelibrary.wiley.com/doi/abs/10.1111/1467-9884.00122`.

[52] B Johnson, Y Song, E Murphy-Hill, and R Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, may 2013. doi: 10.1109/ICSE.2013.6606613.

[53] C. F. Kemerer and M. C. Paulk. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Transactions on Software Engineering*, 35(4):534–550, July 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.27.

[54] J Kim, Y K Malaiya, and I Ray. Vulnerability Discovery in Multi-Version Software Systems. In *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pages 141–148, nov 2007. doi: 10.1109/HASE.2007.55.

[55] Barbara Kitchenham and Stuart Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University, 2007.

[56] Ted Kremenek and Dawson Engler. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In Radhia Cousot, editor, *Static Analysis*, pages 295–315, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-44898-3.

[57] Ivan Victor Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, May 1998.

[58] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a Social Network or a News Media? In *Proc. 19th Int. Conf. on World Wide Web*, pages 591–600, New York, NY, USA, 2010. Association of Computing Machinery. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772751. URL `http://doi.acm.org/10.1145/1772690.1772751`.

[59] Himabindu Lakkaraju, Stephen H. Bach, and Jure Leskovec. Interpretable Decision Sets: A Joint Framework for Description and Prediction. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 1675–1684, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939874. URL `http://doi.acm.org/10.1145/2939672.2939874`.

[60] J.R. Landis and G.G. Koch. The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159–174, 1977. ISSN 0006341X, 15410420. URL `http://www.jstor.org/stable/2529310`.

[61] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-39538-6. doi: 10.1007/3-540-39538-5. URL `https://doi.org/10.1007/3-540-39538-5`.

[62] L Layman, L Williams, and R S Amant. Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 176–185, sep 2007. doi: 10.1109/ESEM.2007.11.

[63] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. Does Bug Prediction Support Human Developers? Findings from a Google Case Study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 372–381, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL `http://dl.acm.org/citation.cfm?id=2486788.2486838`.

[64] Ondvrej Lhoták. Comparing Call Graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 37–42, New York, NY, USA, 2007. Association of Computing Machinery.

ISBN 978-1-59593-595-3. doi: 10.1145/1251535.1251542. URL http://doi.org/10.1145/1251535.1251542.

[65] Peng Li and Baojiang Cui. A Comparative Study on Software Vulnerability Static Analysis Techniques and Tools. In *2010 IEEE International Conference on Information Theory and Information Security*, pages 521–524, Dec 2010. doi: 10.1109/ICITIS.2010.5689543.

[66] S. Lipner. The Trustworthy Computing Security Development Lifecycle. In *Proc. 20th Computer Security Applications Conference*, pages 2–13, Dec 2004. doi: 10.1109/CSAC.2004.41.

[67] B Liu, L Shi, Z Cai, and M Li. Software Vulnerability Discovery Techniques: A Survey. In *2012 Fourth International Conference on Multimedia Information Networking and Security*, pages 152–156, nov 2012. doi: 10.1109/MINES.2012.202.

[68] V Benjamin Livshits and Monica S Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

[69] G. Macbeth, E. Razumiejczyk, and R.D. Ledesma. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10:545 – 555, 05 2011. ISSN 1657-9267. URL http://www.scielo.org.co/scielo.php?script=sci_arttext&pid=S1657-92672011000200018&nrm=iso.

[70] L. MacLeod, M. Greiler, M.A. Storey, C. Bird, and J. Czerwonka. Code Reviewing in the Trenches: Understanding Challenges and Best Practices. *IEEE Software*, PP(99): 1, 2017. ISSN 0740-7459. doi: 10.1109/MS.2017.265100500.

[71] P K Manadhata and J M Wing. An Attack Surface Metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, may 2011. ISSN 2326-3881. doi: 10.1109/TSE.2010.60.

[72] Pratyusa Manadhata, Jeannette Wing, Mark Flynn, and Miles McQueen. Measuring the Attack Surfaces of Two FTP Daemons. In *Proc. 2nd Workshop Quality of Protection*, pages 3–10, New York, NY, USA, 2006. Association of Computing Machinery. ISBN 1-59593-553-3. doi: 10.1145/1179494.1179497. URL http://doi.acm.org/10.1145/1179494.1179497.

[73] Pratyusa K. Manadhata. *An Attack Surface Metric*. PhD thesis, School of Computer Science, Nov 2008. URL http://reports-archive.adm.cs.cmu.edu/anon/2008/CMU-CS-08-152.pdf.

[74] Pratyusa K. Manadhata, Yuecel Karabulut, and Jeannette M. Wing. Report: Measuring the Attack Surfaces of Enterprise Software. In Fabio Massacci, Samuel T. Redwine, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, pages 91–100, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-00199-4.

[75] Steve Mansfield-Devine. Threat hunting: assuming the worst to strengthen resilience. *Network Security*, 2017(5):13 – 17, 2017. ISSN 1353-4858. doi: https://doi.org/10.1016/S1353-4858(17)30050-8. URL http://www.sciencedirect.com/science/article/pii/S1353485817300508.

[76] Paolo Massa and Paolo Avesani. Trust-Aware Recommender Systems. In *Proceedings of the 2007 ACM Conference on Recommender Systems*, RecSys '07, page 17–24, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937308. doi: 10.1145/1297231.1297235. URL https://doi.org/10.1145/1297231.1297235.

[77] Fabio Massacci and Viet Hung Nguyen. Which is the Right Source for Vulnerability Studies?: An Empirical Analysis on Mozilla Firefox. In *Proc. 6th Int. Workshop Security Measurements and Metrics*, MetriSec '10, pages 4:1–4:8, New York, NY, USA, 2010. Association of Computing Machinery. ISBN 978-1-4503-0340-8. doi: 10.1145/1853919.1853925. URL http://doi.acm.org/10.1145/1853919.1853925.

[78] D. McKinney. Vulnerability Bazaar. *IEEE Security & Privacy*, 5(6):69–73, Nov 2007. ISSN 1540-7993. doi: 10.1109/MSP.2007.180.

[79] Vaibhav Mehta, Constantinos Bartzis, Haifeng Zhu, Edmund Clarke, and Jeannette Wing. Ranking Attack Graphs. In Diego Zamboni and Christopher Kruegel, editors, *Recent Advances in Intrusion Detection*, pages 127–144, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-39725-0. URL http://dx.doi.org/10.1007/11856214_7.

[80] P. Mell, K. Scarfone, and S. Romanosky. Common Vulnerability Scoring System. *IEEE Security Privacy*, 4(6):85–89, Nov 2006. ISSN 1558-4046. doi: 10.1109/MSP.2006.145.

[81] Andrew Meneely and Laurie Williams. Secure Open Source Collaboration: An Empirical Study of Linus' Law. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 453–462, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653717. URL http://doi.acm.org/10.1145/1653662.1653717.

[82] Andrew Meneely and Laurie Williams. Strengthening the Empirical Analysis of the Relationship Between Linus' Law and Software Security. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 9:1–9:10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0039-1. doi: 10.1145/1852786.1852798. URL http://doi.acm.org/10.1145/1852786.1852798.

[83] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. Predicting Failures with Developer Networks and Social Network Analysis. In *Proc. 16th Int. Symp. Foundations of Software Engineering*, pages 13–23, New York, NY, USA, 2008. Association of Computing Machinery. ISBN 978-1-59593-995-1. doi: 10.1145/1453101.1453106. URL http://doi.acm.org/10.1145/1453101.1453106.

[84] Andrew Meneely, Ben Smith, and Laurie Williams. Validating Software Metrics: A Spectrum of Philosophies. *ACM Trans. Softw. Eng. Methodol.*, 21(4):24:1—-24:28, feb 2013. ISSN 1049-331X. doi: 10.1145/2377656.2377661. URL http://doi.acm.org/10.1145/2377656.2377661.

[85] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. *International Symposium on Empirical Software Engineering and Measurement*, pages 65–74, 2013. ISSN 19493770. doi: 10.1109/ESEM.2013.19.

[86] Andrew Meneely, Alberto C. Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. An Empirical Investigation of Socio-Technical Code Review Metrics and Security Vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, SSE 2014, page 37–44, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332279. doi: 10.1145/2661685.2661687. URL https://doi.org/10.1145/2661685.2661687.

[87] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors' ". *IEEE Transactions on Software Engineering*, 33(9):637–640, Sept 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.70721.

[88] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, Dec 2010. ISSN 1573-7535. doi: 10.1007/s10515-010-0069-5. URL https://doi.org/10.1007/s10515-010-0069-5.

[89] G. Michael and L. Williams. Toward the Use of Automated Static Analysis Alerts for Early Identification of Vulnerability- and Attack-prone Components. In *Second International Conference on Internet Monitoring and Protection (ICIMP 2007)*, pages 18–18, July 2007. ISBN 0769529119. doi: 10.1109/ICIMP.2007.46.

[90] Christoph Molnar. *Interpretable Machine Learning*. Leanpub, 2019. https://christophm.github.io/interpretable-ml-book/.

[91] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. Challenges with Applying Vulnerability Prediction Models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, HotSoS '15, pages 4:1—-4:9, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3376-4. doi: 10.1145/2746194.2746198. URL http://doi.acm.org/10.1145/2746194.2746198.

[92] Patrick Morrison, David Moye, Rahul Pandita, and Laurie Williams. Mapping the field of software life cycle security metrics. *Information and Software Technology*, 102:146–159, 2018. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.05.011. URL http://www.sciencedirect.com/science/article/pii/S095058491830096X.

[93] Patrick J Morrison, Rahul Pandita, Xusheng Xiao, Ram Chillarege, and Laurie Williams. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering*, Sept 2017. ISSN 1573-7616. doi: 10.1007/s10664-017-9541-1. URL https://doi.org/10.1007/s10664-017-9541-1.

[94] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. Using complexity metrics to improve software security. *Computer Fraud & Security*, 2013(5):8 – 17, 2013. ISSN 1361-3723. doi: https://doi.org/10.1016/S1361-3723(13)70045-9. URL http://www.sciencedirect.com/science/article/pii/S1361372313700459.

[95] N. Munaiah, A. Meneely, and P. K. Murukannaiah. A Domain-Independent Model for Identifying Security Requirements. In *2017 IEEE 25th International Requirements Engineering Conference (RE)*, pages 506–511, Sept 2017. doi: 10.1109/RE.2017.79.

[96] N. Munaiah, A. Rahman, J. Pelletier, L. Williams, and A. Meneely. Characterizing Attacker Behavior in a Cybersecurity Penetration Testing Competition. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, Sep 2019. doi: 10.1109/ESEM.2019.8870147.

[97] Nuthan Munaiah. Assisted Discovery of Software Vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, ICSE '18, pages 464–467, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5663-3. doi: 10.1145/3183440.3183453. URL `http://doi.acm.org/10.1145/3183440.3183453`.

[98] Nuthan Munaiah. `archeogit`. `https://github.com/samaritan/archeogit/releases`, Dec 2019. Version: 0.1.1-alpha.

[99] Nuthan Munaiah and Andrew Meneely. Attack Surface Meter. `https://github.com/andymeneely/attack-surface-metrics`, Aug 2016. Version: 0.11.0.

[100] Nuthan Munaiah and Andrew Meneely. Beyond the attack surface: Assessing security risk with random walks on call graphs. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, pages 3–14, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4576-7. doi: 10.1145/2995306.2995311. URL `http://doi.acm.org/10.1145/2995306.2995311`.

[101] Nuthan Munaiah and Andrew Meneely. Vulnerability Severity Scoring and Bounties: Why the Disconnect? In *Proceedings of the 2Nd International Workshop on Software Analytics*, SWAN 2016, pages 8–14, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4395-4. doi: 10.1145/2989238.2989239. URL `http://doi.acm.org/10.1145/2989238.2989239`.

[102] Nuthan Munaiah and Andrew Meneely. Data-driven Insights from Vulnerability Discovery Metrics. In *Proceedings of the Joint 4th International Workshop on Rapid Continuous Software Engineering and 1st International Workshop on Data-Driven Decisions, Experimentation and Evolution*, RCoSE-DDrEE '19, pages 1–7, Piscataway, NJ, USA, 2019. IEEE Press. doi: 10.1109/RCoSE/DDrEE.2019.00008. URL `https://doi.org/10.1109/RCoSE/DDrEE.2019.00008`.

[103] Nuthan Munaiah and Andrew Meneely. Data Set from the Systematization of Vulnerability Discovery Metrics, Mar 2020. URL `https://doi.org/10.5281/zenodo.3700824`.

[104] Nuthan Munaiah, Felivel Camilo, Wesley Wigham, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project. *Empirical Software Engineering*, 22(3):1305–1347, Jun 2017. ISSN 1573-7616. doi: 10.1007/s10664-016-9447-3. URL `https://doi.org/10.1007/s10664-016-9447-3`.

[105] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating GitHub for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, Dec 2017. ISSN 1573-7616. doi: 10.1007/s10664-017-9512-6. URL `https://doi.org/10.1007/s10664-017-9512-6`.

[106] Nuthan Munaiah, Benjamin S Meyers, Cecilia O Alm, Andrew Meneely, Pradeep K Murukannaiah, Emily Prud'hommeaux, Josephine Wolff, and Yang Yu. Natural Language Insights from Code Reviews that Missed a Vulnerability. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, pages 70–86. Springer International Publishing, Cham, 2017. ISBN 978-3-319-62105-0. doi: 10.1007/978-3-319-62105-0_5. URL `https://doi.org/10.1007/978-3-319-62105-0{_}5`.

[107] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An Empirical Study of Static Call Graph Extractors. *Transactions on Software Engineering and Methodology*, 7(2):158–191, Apr 1998. ISSN 1049-331X. doi: 10.1145/279310.279314. URL `http://doi.org/10.1145/279310.279314`.

[108] Nachiappan Nagappan and Thomas Ball. Static Analysis Tools As Early Indicators of Pre-release Defect Density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 580–586, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2. doi: 10.1145/1062455.1062558. URL `http://doi.acm.org/10.1145/1062455.1062558`.

[109] Stephan Neuhaus and Thomas Zimmermann. The Beauty and the Beast: Vulnerabilities in Red Hat's Packages. In *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC)*, USENIX ATC '09, 2009. URL `https://www.usenix.org/legacy/event/usenix09/tech/full{_}papers/neuhaus/neuhaus{_}html/`.

[110] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. *Proceedings of the 14th ACM conference on Computer and communications security CCS 07*, page 529, 2007. ISSN 15437221. doi: 10.1145/1315245.1315311. URL `http://portal.acm.org/citation.cfm?doid=1315245.1315311`.

[111] Viet Hung Nguyen and Le Minh Sang Tran. Predicting Vulnerable Software Components with Dependency Graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pages 3:1–3:8, New York, NY, USA, 2010. Association of Computing Machinery. ISBN 9781450303408. doi: 10.1145/1853919.1853923. URL `http://doi.org/10.1145/1853919.1853923`.

[112] Andy Ozment. Improving Vulnerability Discovery Models. In *Proceedings of the 2007 ACM Workshop on Quality of Protection*, QoP '07, pages 6–11, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-885-5. doi: 10.1145/1314257.1314261. URL `http://doi.acm.org/10.1145/1314257.1314261`.

[113] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, Nov 1999. URL `http://ilpubs.stanford.edu:8090/422/`. Previous number = SIDL-WP-1999-0120.

[114] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[115] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 426–437, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813604. URL `http://doi.acm.org/10.1145/2810103.2813604`.

[116] Martin Pinzger, Nachiappan Nagappan, and Brendan Murphy. Can Developer-Module Networks Predict Failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 2–12, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595939951. doi: 10.1145/1453101.1453105. URL `https://doi.org/10.1145/1453101.1453105`.

[117] B. Potter and G. McGraw. Software Security Testing. *IEEE Security & Privacy*, 2(5): 81–85, Sept 2004. ISSN 1540-7993. doi: 10.1109/MSP.2004.84.

[118] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015. URL `http://www.R-project.org/`.

[119] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018. URL `https://www.R-project.org/`.

[120] Foyzur Rahman and Premkumar Devanbu. How, and Why, Process Metrics Are Better. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 432–441, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL `http://dl.acm.org/citation.cfm?id=2486788.2486846`.

[121] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, volume 1. O'Reilly Media, 1999. ISBN 9781565927247.

[122] Rochester Institute of Technology. Research Computing Services, 2019. URL `https://www.rit.edu/researchcomputing/`.

[123] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M. González-Barahona. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. *Information and Software Technology*, 99:164 – 176, 2018. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2018.03.009. URL `http://www.sciencedirect.com/science/article/pii/S0950584917304275`.

[124] J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys. In *Annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[125] A. Sabetta and M. Bezzi. A Practical Approach to the Automatic Classification of Security-Relevant Commits. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 579–582, Sep 2018. doi: 10.1109/ICSME.2018.00058.

[126] C Sadowski, J v. Gogh, C Jaspan, E Söderberg, and C Winter. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608, may 2015. doi: 10.1109/ICSE.2015.76.

[127] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting Vulnerable Software Components via Text Mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, Oct 2014. ISSN 0098-5589. doi: 10.1109/TSE.2014.2340398.

[128] Secure Software Engineers. Rough Auditing Tool for Security (RATS). `https://code.google.com/archive/p/rough-auditing-tool-for-security/`, Dec 2013. Version: 2.4.

[129] C. Severance. Bruce Schneier: The Security Mindset. *Computer*, 49(2):7–8, Feb 2016. ISSN 1558-0814. doi: 10.1109/MC.2016.38.

[130] Hossain Shahriar and Mohammad Zulkernine. Mitigating Program Security Vulnerabilities: Approaches and Challenges. *ACM Comput. Surv.*, 44(3):11:1—-11:46, jun 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187673. URL `http://doi.acm.org/10.1145/2187671.2187673`.

[131] Lwin Khin Shar and Hee Beng Kuan Tan. Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10):1767–1780, 2013. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2013.04.002. URL `http://www.sciencedirect.com/science/article/pii/S0950584913000852`.

[132] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 273–284, May 2002. doi: 10.1109/SECPRI.2002.1004377.

[133] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, Nov 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.81.

[134] Yonghee Shin. *Investigating Complexity Metrics As Indicators of Software Vulnerability*. PhD thesis, North Carolina State University, 2011. AAI3442705.

[135] Yonghee Shin and Laurie Williams. Is Complexity Really the Enemy of Software Security? In *Proceedings of the 4th ACM Workshop on Quality of Protection*, QoP '08, pages 47–50, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-321-1. doi: 10.1145/1456362.1456372. URL `http://doi.acm.org/10.1145/1456362.1456372`.

[136] Yonghee Shin and Laurie Williams. An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 315–317, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-971-5. doi: 10.1145/1414004.1414065. URL `http://doi.acm.org/10.1145/1414004.1414065`.

[137] Yonghee Shin and Laurie Williams. An Initial Study on the Use of Execution Complexity Metrics As Indicators of Software Vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, SESS '11, pages 1–7, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0581-5. doi: 10.1145/1988630.1988632. URL `http://doi.acm.org/10.1145/1988630.1988632`.

[138] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, Feb 2013. ISSN 1573-7616. doi: 10.1007/s10664-011-9190-8. URL `https://doi.org/10.1007/s10664-011-9190-8`.

[139] Miltiadis Siavvas, Erol Gelenbe, Dionysios Kehagias, and Dimitrios Tzovaras. Static Analysis-Based Approaches for Secure Software Development. In Erol Gelenbe, Paolo Campegiani, Tadeusz Czachórski, Sokratis K. Katsikas, Ioannis Komnios, Luigi Romano, and Dimitrios Tzovaras, editors, *Security in Computer and Information Sciences*, pages 142–157, Cham, 2018. Springer International Publishing. ISBN 978-3-319-95189-8.

[140] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When Do Changes Induce Fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005. ISSN 0163-5948. doi: 10.1145/1082983.1083147. URL `http://doi.acm.org/10.1145/1082983.1083147`.

[141] B. Smith and L. Williams. Using SQL Hotspots in a Prioritization Heuristic for Detecting All Types of Web Application Vulnerabilities. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 220–229, March 2011. doi: 10.1109/ICST.2011.15.

[142] Sarah E. Smith, Laurie Williams, and Jun Xu. Expediting Programmer AWAREness of Anomalous Code. In *2005 IEEE 16th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–2, Nov 2005.

[143] Margaret-Anne Storey and Alexey Zagalsky. Disrupting Developer Productivity One Bot at a Time. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 928–931, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2983989. URL `http://doi.acm.org/10.1145/2950290.2983989`.

[144] SurveyMonkey. Sample Size Calculator: Understanding Sample Sizes | SurveyMonkey. `https://www.surveymonkey.com/mp/sample-size-calculator/`, Jan 2020. Accessed: 2020-01-22.

[145] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams. Approximating Attack Surfaces with Stack Traces. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2 of *ICSE '15*, pages 199–208, May 2015. doi: 10.1109/ICSE.2015.148.

[146] Christopher R. Theisen. *Risk-Based Attack Surface Approximation*. PhD thesis, North Carolina State University, 2018.

[147] James J. Treinen and Ramakrishna Thurimella. Application of the PageRank Algorithm to Alarm Graphs. In Sihan Qing, Hideki Imai, and Guilin Wang, editors,

*Information and Communications Security*, pages 480–494, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-77048-0. doi: 10.1007/978-3-540-77048-0_37. URL http://dx.doi.org/10.1007/978-3-540-77048-0_37.

[148] J. Walden and M. Doyle. SAVI: Static-Analysis Vulnerability Indicator. *IEEE Security & Privacy*, 10(3):32–39, May 2012. ISSN 1540-7993. doi: 10.1109/MSP.2012.1.

[149] J. Walden, J. Stuckman, and R. Scandariato. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 23–33, Nov 2014. doi: 10.1109/ISSRE.2014.32.

[150] Elaine J. Weyuker, Thomas J. Ostrand, and Robert M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008. ISSN 13823256. doi: 10.1007/s10664-008-9082-8.

[151] David A. Wheeler. Flawfinder. https://dwheeler.com/flawfinder/flawfinder.pdf, Oct 2019. Version: 293ca1.

[152] Jim Whitehead. Collaboration in Software Engineering: A Roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 214–225, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.4. URL http://dx.doi.org/10.1109/FOSE.2007.4.

[153] Rebecca S Wills. Google's PageRank: The Math Behind the Search Engine. *The Mathematical Intelligencer*, 28(4):6–11, 2008. ISSN 0343-6993. doi: 10.1007/BF02984696. URL http://dx.doi.org/10.1007/BF02984696.

[154] A. A. Younis and Y. K. Malaiya. Using Software Structure to Predict Vulnerability Exploitation Potential. In *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, pages 13–18, June 2014. doi: 10.1109/SERE-C.2014.17.

[155] A. A. Younis, Y. K. Malaiya, and I. Ray. Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 1–8. IEEE, Jan 2014. doi: 10.1109/HASE.2014.10.

[156] Awad Younis, Yashwant Malaiya, Charles Anderson, and Indrajit Ray. To Fear or Not to Fear That is the Question: Code Characteristics of a Vulnerable Function with an Existing Exploit. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, CODASPY '16, pages 97–104, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3935-3. doi: 10.1145/2857705.2857750. URL http://doi.acm.org/10.1145/2857705.2857750.

[157] Matt Yule-Bennett. Nameko - A microservice framework for Python. https://www.nameko.io, 2020. Version: 2.12.0.

[158] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan. How Does Context Affect the Distribution of Software Maintainability Metrics? In *2013 IEEE International Conference on Software Maintenance*, pages 350–359, Sep. 2013. doi: 10.1109/ICSM.2013.46.

[159] He Zhang, Muhammad Ali Babar, and Paolo Tell. Identifying relevant studies in software engineering. *Information and Software Technology*, 53(6):625–637, 2011. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2010.12.010. URL `http://www.sciencedirect.com/science/article/pii/S0950584910002260`.

[160] T Zimmermann, N Nagappan, and L Williams. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428, Apr 2010. doi: 10.1109/ICST.2010.32.

[161] Thomas Zimmermann and Nachiappan Nagappan. Predicting Defects Using Network Analysis on Dependency Graphs. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 531–540, New York, NY, USA, 2008. Association of Computing Machinery. ISBN 978-1-60558-079-1. doi: 10.1145/1368088.1368161. URL `http://doi.org/10.1145/1368088.1368161`.

[162] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project Defect Prediction: A Large Scale Experiment on Data vs. Domain vs. Process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-001-2. doi: 10.1145/1595696.1595713. URL `http://doi.acm.org/10.1145/1595696.1595713`.

# Appendices

# Appendix A

# SAMARITAN Metrics Platform

In the sections that follow, we introduce the SAMARITAN metrics platform.

## A.1 Metrics

We collected 18 metrics known, from existing literature, to be associated with historical vulnerabilities. The choice of the metrics was informed by the systematic literature review on vulnerability discovery metrics being conducted reported in Chapter 5. The metrics are defined at various levels of granularity, such as function, file, developer, change, and commit. While function, file, and developer levels of granularity need no further elaboration, the commit and change levels of granularity could use elaboration to aid the comprehension of metrics defined at these levels. A commit, in the context of a source code repository, is a unit of change which is always associated with a developer and almost always associated with one or more files. A metric (say, number of files changed) defined at the commit level thus quantifies an attribute of the commit itself. A change, on the other hand, is a subunit of a commit that refers to a specific file changed in a commit. A metric (say, number of lines inserted and deleted) defined at the change level thus quantifies an attribute of a change within a commit. The entity–relationship diagram shown in Figure A.1 depicts the various entities of a software repository (i.e. function, file, developer, change, and commit) and the relationships between them. Each entity has two sets of data associated with it: metadata (shown in black in Figure A.1) to identify the entity and metrics (shown in color in Figure A.1) collected from the entity. The relationships, represented by solid lines between entities, depict the cardinality of the relationship. While some metrics (shown in blue) in Figure A.1) are directly collected from an entity, others (shown in orange in Figure A.1) are aggregated from related entities. In case of aggregated metrics, the directed dotted arrow show the entity from which the metric is aggregated and the aggregation function used.

The 18 metrics available on the SAMARITAN metrics platform:

1. *Collaboration (Centrality)* - The maximum of the edge centrality of edges representing files in a collaboration network [81]. A collaboration network is an unweighted and undirected graph in which nodes represent developers and edges represent files. An edge exists between two developers if they both changed at least one file.

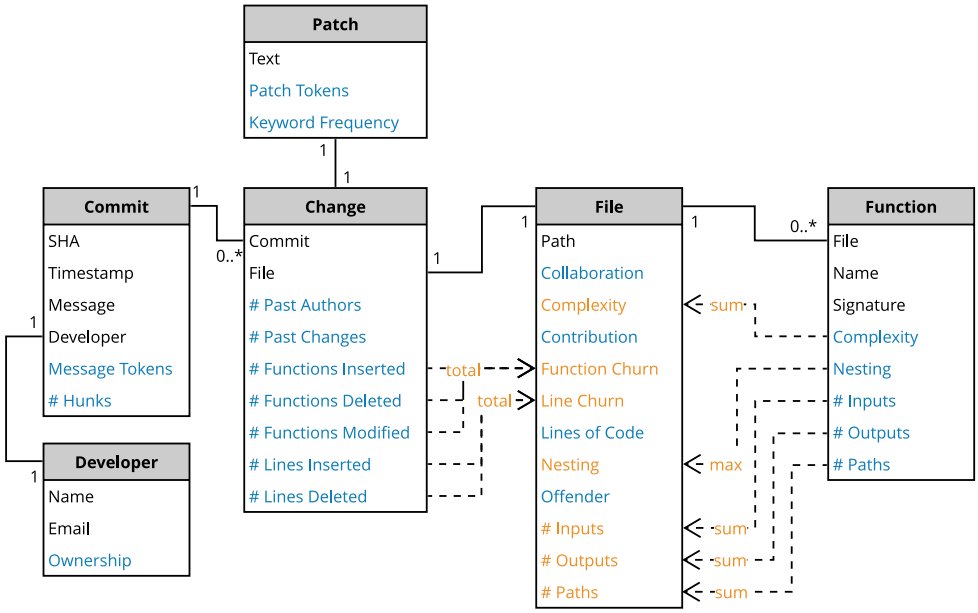



Figure A.1: Entity–relationship diagram depicting the various entities (function, file, developer, change, and commit) from which metrics in the SAMARITAN metrics platform are collected and the relationships between the various entities

2. *Contribution (Centrality)* - The node betweenness centrality of nodes representing files in a contribution network [81]. A contribution network is a weighted and undirected bipartite graph with two sets of nodes: files and developers. An edge exists between a developer node and a file node if the developer made a change (commit) to the file. The weight of the edge is the number of changes a single developer made to a particular file.
3. *(Cyclomatic) Complexity* - The number of unique decision paths through a function [156]. We collected this metric at the function level and aggregated it to the file level by computing the sum of the metric.
4. *Function Churn* - The total number of functions inserted, deleted, and modified by a change [115]. We collected the metric at the change level and aggregated it at the file level by computing the sum of the metric.
5. *(Known) Offender* - A binary-valued metric that indicates if a file has been fixed for a vulnerability in the past [85].
6. *Line Churn* - The number of lines inserted and deleted by a change [160]. We collected this metric at the change level and aggregated it to the file level by computing the sum of the metric.

7. *Message Tokens* - The tokens in a commit message represented using a high dimension vector [115]. A message token in is essentially a word in the commit message.
8. *Nesting* - The maximum nesting level of control structures in a function [156]. We collected this metric at the function level and aggregated it to the file level by computing the maximum of the metric.
9. *Ownership* - The proportion of commits contributed by a developer in a project expressed as a percentage of the total number of commits in the project [115].
10. *Patch Token Existence* - The tokens in a commit patch represented using a high dimension vector [115]. A patch token is a essentially a word in the commit patch with no special programming language preprocessing performed on the patch.
11. *Patch Keyword Frequency* - The frequency of keywords in a commit patch [115]. The keywords are specific to a programming language. We implemented this metric for patches in C, C++, Java, and Python.
12. *Source Lines of Code* - The total number of lines of source code in a file [160].
13. *# Hunks* - The number of hunks (a continuous block of changes to a file) in a commit [115].
14. *# Inputs* - The number of inputs that a function uses [156]. We collected this metric at the function level and aggregated it to the file level by computing the sum of the metric.
15. *# Outputs* - The number of functions that a given function calls [156]. We collected this metric at the function level and aggregated it to the file level by computing the sum of the metric.
16. *# Paths* - The number of unique decision paths through a function [156]. We collected this metric at the function level and aggregated it to the file level by computing the sum of the metric.
17. *# Past Authors* - The number of distinct developers who have contributed changes to a file in the past [115].
18. *# Past Changes* - The number of changes that a file has been subject to in the past [115].

We only collected the metrics from file paths ending with the extensions `.c`, `.cc`, `.cpp`, `.cxx`, `.h`, `.hh`, `.hpp`, `.hxx`, or `.inl` for C/C++ projects and `.java` for Java projects. While we implemented the algorithms to collect the contribution centrality, collaboration centrality, and function churn metrics, we used `git` to collect line churn, ownership, # hunks, # past authors, and # past changes metrics, a combination of `git` and `CountVectorizer` from `scikit-learn` [114] to collect message tokens, patch tokens, and patch keyword frequency metrics, a manual approach to collect the offender metric, and SciTools Understand to collect the remaining metrics.

The implementation of the SAMARITAN metric services is open source with the source code available on GitHub at `https://github.com/samaritan/services/` and Docker images available on GitHub Package Registry at `https://github.com/samaritan/services/packages`. The decision to release the implementation of the metrics as containers is our contribution toward alleviating a challenge for researchers to include the metrics in replication studies and for practitioners to use the metrics in practice.

## A.2 Architecture

The 18 metrics defined in the previous section have been implemented as containerized microservices. We chose to implement the metrics as containerized microservices to allow researchers and practitioners to collect metrics from any project with minimal effort. We hope the convenience lowers the barrier to entry to the use of metrics so much so that researchers are encouraged to consider replicating the metrics and practitioners are encouraged to use the metrics in practice.

Shown in Figure A.2 is the architectural layout of the SAMARITAN metrics platform.

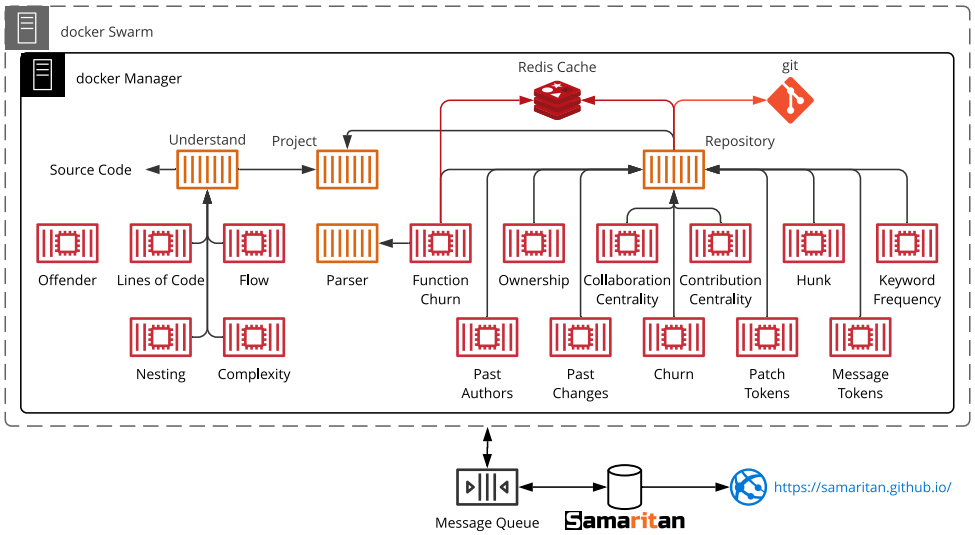



Figure A.2: Architectural layout of the SAMARITAN metrics platform

The metric services are implemented using Nameko, a microservice framework for Python [157]. The containers running the metric services are depicted using red rectangular blocks and containers running the supporting services are depicted using orange rectangular blocks. Although we ran the containers using `docker swarm`, we only had a single Docker machine which acted as both the manager and the worker node. While we only show the message queue being used for communication between SAMARITAN database and the swarm of services, all communications between the services is achieved through the message queue. We do not show the message queue for communications shown inside the manager node for simplicity. The SAMARITAN website, accessible at `https://samaritan.github.io/`, is part of the platform to disseminate the metrics to the community. We summarize the empirical knowledge surrounding a metric in the respective metric page on the website. As we introduce additional metric services to the SAMARITAN platform, the website will get updated to disseminate the metrics' knowledge to the community.

# Appendix B

# Parameter Tuning

In this section, we describe the methodology used to tune the parameters (i.e. personalization vector, damping factor, and edge weights vector) for the risky walk metric. The objective is to explore parameter values that, when used to compute the risky walk metric, enables a clear delineation of functions that were fixed for a historical post-release vulnerability (i.e. historically vulnerable functions/files) from those that were not (i.e. neutral functions/files). Although the label "historically vulnerable" may seem similar to the term "vulnerable" introduced in Section 4.4.3, there is a key difference in usage. To understand the difference, consider a sequence of chronologically ordered FFmpeg releases with version numbers 1.0.0, 1.1.0, 1.2.0, and 1.2.1. All functions/files fixed for a vulnerability in 1.0.0 and 1.1.0 are considered *historically vulnerable* in 1.1.0, whereas, all functions/files fixed for a vulnerability in 1.2.1 are considered *vulnerable* in 1.2.0. Furthermore, the set of vulnerable functions/files and the set of historically vulnerable functions/files do not intersect in any given release.

Our overall approach for parameter tuning is a brute-force exploration of parameter values along an exponential scale. Collectively, we examined the following seven variables:

> Damping factor ($\alpha$), personalization of entry points ($P_{entry}$), personalization of exit points ($P_{exit}$), weight of call edges ($W_{call}$), weight of return edges ($W_{return}$), additive weight for edges terminating at dangerous points ($Aw_{dangerous}$), and additive weight for edges terminating at historically vulnerable functions/files ($Aw_{vulnerable}$)

For the personalization vector, the attack surface metaphor argument says that an attacker is likely to start the reconnaissance for an attack at either an entry point or an exit point (e.g. via fuzz testing techniques). We capture this behavior by having the personalization vector contain *higher probability* for entry and exit points than non-entry and non-exit points. For the non-entry and non-exit points, the personalization vector contains probability drawn from a uniform distribution.

For the damping factor, tailoring the input to the system is the only way for an attacker to affect the flow of control through the call graph. We could conceive situations where explorations would end quickly or after a long time, so in our parameter tuning we considered a wide range of values.

For assigning weights to edges, we considered four types of edges: calls, returns, edges to dangerous points, and edges to historically vulnerable functions/files. We assigned all call and return edges a base weight. A function/file that makes dangerous system calls or was historically vulnerable could be potential targets for an attacker. We capture this behavior by increasing the weight of edges terminating at such functions/files. Weights for edges become probabilities by summing them per node and dividing each by the total for that node.

We defined a set of candidate values for each of the seven variables and constructed a collection of 7-tuple permutations of the candidate values. The range of candidate values for each of the seven variables were:(a) $\alpha$ from 0.1 to 0.9, (b) $P_{entry}$ and $P_{exit}$ from 1 to 1,000,000, (c) $W_{call}$ and $W_{return}$ from 10 to 10,000, and (d) $Aw_{dangerous}$ and $Aw_{vulnerable}$ from 10 to 1,000. While the candidate values for $\alpha$ were on a linear scale (with an interval of 0.1), the candidate values for the remaining variables were on an exponential scale. Although the candidate values for the personalization variables ($P_{entry}$ and $P_{exit}$) are not probabilities, they are transformed into probabilities before being used in the algorithm.

The total number of permutations in the collection was 63,504. We used an iterative approach to evaluate each permutation to obtain a set of values for the PageRank parameters, which when used in the computation of the risky walk metric, results in the metric being statistically significantly associated (p-value $\leq 0.05$) with historically vulnerable functions and have the largest effect size evaluation. We used the non-parametric Mann-Whitney-Wilcoxon (MWW) test to assess the association and Cohen's $d$ effect size statistic [24] to assess the effect size. The process was repeated in all 16 releases of FFmpeg and 7 releases of Wireshark. The permutation that resulted in risky walk having the largest average value of Cohen's $d$ when aggregated across releases in each subject was chosen to compose the PageRank parameters. The highest ranking value of the parameters in FFmpeg and Wireshark is presented in Table B.1.

Table B.1: Highest ranking value of the variables that compose the PageRank parameters in FFmpeg and Wireshark

| **Variable** | **Subject** | |
|---|---|---|
| | **FFmpeg** | **Wireshark** |
| $\alpha$ | 0.9 | 0.9 |
| $P_{entry}$ | 1 | 10,000 |
| $P_{exit}$ | 1 | 10,000 |
| $W_{call}$ | 10 | 100 |
| $W_{return}$ | 10 | 10 |
| $Aw_{dangerous}$ | 10 | 10 |
| $Aw_{vulnerable}$ | 1,000 | 1,000 |

To avoid over-fitting the parameters, we also ran a sensitivity analysis of our prediction question (RQ2, in Section 4.5.2) by using the average of the parameter values of the top 100 highest ranking permutations ordered by the average Cohen's $d$ (aggregated across releases).

The final precision and recall was within 2.88% of those obtained from the model with the averaged weights.

# Appendix C

# Quasi-Gold Standard Set

The primary studies listed below compose the Quasi-Gold Standard set that was used to validate the search string in accordance with the Quasi-Gold Standard approach prescribed by Zhang *et al.* [159]. The primary studies in the Quasi-Gold Standard set were curated based on personal experience of the authors in conducting empirical research in vulnerability discovery metrics. The evidence of this fact is in five of the primary studies in the Quasi-Gold Standard set being authored by at least one of the authors of this systematic review.

QGS1 Predicting Vulnerable Software Components [110]

QGS2 An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics [136]

QGS3 Secure Open Source Collaboration: An Empirical Study of Linus' Law [81].

QGS4 The Beauty and the Beast: Vulnerabilities in Red Hat's Packages [109]

QGS5 Strengthening the Empirical Analysis of the Relationship Between Linus' Law and Software Security [82]

QGS6 Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista [160]

QGS7 Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities [133]

QGS8 An Initial Study on the Use of Execution Complexity Metrics As Indicators of Software Vulnerabilities [137]

QGS9 Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities [22]

QGS10 Can traditional fault prediction models be used for vulnerability prediction? [138]

QGS11 When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits [85]

QGS12 Predicting Vulnerable Software Components via Text Mining [127]

QGS13 Predicting Vulnerable Components: Software Metrics vs Text Mining [149]

QGS14 VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits [115]

QGS15 To Fear or Not to Fear That is the Question: Code Characteristics of a Vulnerable Function with an Existing Exploit [156]

QGS16 Beyond the Attack Surface: Assessing Security Risk with Random Walks on Call Graphs [100]

QGS17 Do bugs foreshadow vulnerabilities? An in-depth study of the Chromium project [104]

QGS18 Natural Language Insights from Code Reviews that Missed a Vulnerability [106]

# Appendix D

# Data Extraction Form

The form that will facilitate the extraction of data from primary studies to address the research questions is shown in Table D.1. The design of the data extraction form was informed by the type of data needed to address the research questions. As a result, the form shown in Table D.1 is logically divided into sections with each section, except the first section (labeled Miscellaneous Metadata), being labeled after the research question being addressed using the data collected through the data extraction fields in the section. We implemented the first two sections of the data extraction form using Airtable[1] to simplify the collection of data. We used Google Sheets to collect data for the third section since it was intuitive to define a matrix associating primary studies with validation criterion. When collecting the data using the Google Sheets sheet, we used "Yes" to indicate a primary study (in a row) subjected metrics to a validation criterion (in a column), "No" otherwise.

Table D.1: Data extraction form used to collect data from primary studies in the systematic literature review of vulnerability discovery metrics

| Item | Data Type | Description |
|---|---|---|
| **Miscellaneous Metadata (For Each Primary Study)** | | |
| Extracted By | Plain Text | The name of the researcher who extracted the data. |
| Verified By | Plain Text | The name of the researcher who verified the extracted data. |
| Identifier | Plain Text | The unique identifier of the primary study from which the data was extracted. |
| # Subjects | Numeric | The number of subjects of study that a metric have been collected from. |

Continued on next page

[1] `https://airtable.com`

Table D.1. Data extraction form used to collect data from primary studies in the systematic literature review of vulnerability discovery metrics (Continued)

| Item | Data Type | Description |
|---|---|---|
| Subjects' Nature | Categorical | The type of subject of study. Open-source and closed-source are the two categories considered. |
| Subjects' Language | Categorical | The primary programming language in which the subject of study is written in. |
| Response Variable | Plain Text | The description of the response variable in the study. In some vulnerability discovery studies, number of vulnerabilities is used as the response variable while others used vulnerableness (i.e. is a function, file, component, or software likely to be vulnerable). The response variable is an important piece of information that must be captured as part of the data extraction. |
| # Vulnerabilities | Numeric | The number of vulnerabilities that were considered in the empirical analysis. |
| Miscellaneous | Varying | Miscellaneous metadata about the primary study such as year of publication, name of the journal/conference/workshop in which the study was included, and number of pages. |
| **RQ 1 - Enumeration (For Each Metric)** | | |
| Name | Plain Text | The name of a vulnerability discovery metric. |
| Definition | Plain Text | The definition of a vulnerability discovery metric. |
| Type | Plain Text | The data type of the vulnerability discovery metric. |
| Applicability | Categorical | The applicability of the metric. For instance, if the number of late-night commits is posited to be related to vulnerability discovery, the applicability of the metric would be categorized as process because the aspect of Software Engineering that the metric applies to is Process (Version Control) |
| Granularity | Plain Text | If applicable, the granularity at which the metric is defined. For instance, if SLOC is posited to be related to vulnerability discovery, was SLOC collected at the function-level, method-level, class-level, file-level, or component-/module-level. |

Continued on next page

Table D.1. Data extraction form used to collect data from primary studies in the systematic literature review of vulnerability discovery metrics (Continued)

| Item | Data Type | Description |
|---|---|---|
| Aggregated | Boolean | In assessing the validity of a metric, was the metric values aggregated to a higher level of granularity than the one at which the metric was collected? For instance, if number of lines added and deleted (i.e. churn) in a commit is a metric posited to be related to vulnerability discovery, then, depending on the level of granularity of the analysis, the metric may have been aggregated to the file-level or component-/module-level prior to analysis. |
| Aggregated From | Plain Text | If applicable, the granularity at which the metric was collected. |
| Aggregated To | Plain Text | If applicable, the granularity at which the metric was aggregated to prior to analysis. |
| **RQ 2 - Validation (For Each Primary Study)** | | |
| Validation Criteria | Categorical | The name of the validation criterion, one that is associated with the benefit of informing decisions, that metrics in a primary subject have been subject to. We use the 10 *atomic* decision-informing validation criteria enumerated by Meneely *et al.* in their systematic review of metric validation criteria [84]. For instance, if SLOC was one of the metrics empirically evaluated to be associated with historical, we say that the authors of the primary study have demonstrated the *#5 Association* validity. |
| Developer Feedback | Boolean | In validating the decision-informing ability of a metric, was feedback from real developers' sought? |

# Appendix E

# Primary Studies

The candidate studies that satisfied the inclusion and exclusion criteria established during the planning stage of the systematic literature review process are enumerated below. These candidates studies are referred to as primary studies. Along with the studies in the Quasi-Gold Standard Set, the primary studies enumerated below form the source of data used to address the research questions.

P1 Vulnerability Discovery in Multi-Version Software Systems [54]

P2 Prioritizing Software Security Fortification Through Code-Level Metrics [37]

P3 Predicting Attack-prone Components [36]

P4 Toward Non-security Failures as a Predictor of Security Faults and Failures [38].

P5 Can Complexity, Coupling, and Cohesion Metrics Be Used as Early Indicators of Vulnerabilities? [20]

P6 Predicting Vulnerable Software Components with Dependency Graphs [111]

P7 An Attack Surface Metric [71]

P8 Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns [131]

# Appendix F

# Vulnerability Discovery Metrics

The vulnerability discovery metrics identified in the systematic literature review are enumerated in Table F.1. The ordering of the enumeration is based on the number of primary studies in which the metric was considered. For instance, source lines of code metric was the most common metric with 14 primary studies in which the metric appeared.

Table F.1: Vulnerability discovery metrics identified in the systematic literature review

| **Metric (# Primary Studies)** | | |
|---|---|---|
| Source Lines of Code (14) | Incoming Information Flow (aka Fan In) (8) | Nesting Degree (8) |
| Cyclomatic Complexity (7) | Outgoing Information Flow (aka Fan Out) (7) | Essential Complexity (6) |
| Strict Cyclomatic Complexity (6) | Number of Commits (5) | Number of Unique Contributors (5) |
| Comment Density (4) | Line Churn (4) | Number of Functions Defined (4) |
| Path Count (4) | Coupling between Classes (3) | Depth of Inheritance Tree (3) |
| Developer Network Edge (File) Betweenness (3) | Modified Cyclomatic Complexity (3) | Number of Declarative Lines of Code (3) |
| Number of Incoming Connections (3) | Number of Outgoing Connections (3) | Weighted Methods per Class (3) |
| Code Token Frequency (2) | Component Import (2) | Contribution Network File Node Betweenness (2) |

Continued on next page

Table F.1. Vulnerability discovery metrics identified in the systematic literature review (Continued)

| Metric (# Primary Studies) | | |
|---|---|---|
| Density of Static Analysis Warnings (2) | Henry-Kafura (2) | Lack of Cohesion of Methods (2) |
| Number of Base Classes (2) | Number of Children (2) | Number of Faults (2) |
| Number of Function Callers (2) | Number of Functions Called (2) | Number of Lines Added (2) |
| Number of Lines Deleted (2) | Number of Lines Modified (2) | Number of New Lines (2) |
| Number of Preprocessor Lines of Code (2) | Relative Line Churn (2) | Response Set for Class (2) |
| Arc Coverage (1) | Attack Surface Measurement (1) | Average External Data Flow (1) |
| Average Incoming Data Flow (1) | Average Internal Data Flow (1) | Average Outgoing Data Flow (1) |
| Baseline Commit (1) | Blank Lines of Code (1) | Block Coverage (1) |
| Build Experience (1) | Comment Lines of Code (1) | Compatability Experience (1) |
| Contribution Network File Node Closeness (1) | Dependency Betweenness (1) | Dependency Eigenvector Centrality (1) |
| Depth of Master Ownership (1) | Developer Network Node (Developer) Betweeness (1) | Developer Network Node (Developer) Closeness (1) |
| Developer Network Node (Developer) Degree (1) | Distance to Kernel (1) | Edit Frequency (1) |
| Exclusive Execution Time (1) | Frazier Score (1) | Function Call (1) |
| Halstead Volume (1) | Inclusive Execution Time (1) | Inquisitiveness (1) |
| Interface Complexity (1) | Interface Edges (1) | Keyword Frequency (1) |
| Known Offender (1) | Level of Organizational Code Ownership (1) | Negativity (1) |
| New Effective Author (1) | Non-text-based Data Access (1) | Notable Commit (1) |
| Number of Affected Authors (1) | Number of Alpha Testing Non-security Faults (1) | Number of Beta Testing Non-security Faults (1) |
| Number of Bugs (1) | Number of Build Bugs (1) | Number of Compatability Bugs (1) |
| Number of Customer-reported Non-security Faults (1) | Number of Data Encoding Nodes (1) | Number of Data Encrypting Nodes (1) |

Continued on next page

Table F.1. Vulnerability discovery metrics identified in the systematic literature review (Continued)

| Metric (# Primary Studies) | | |
|---|---|---|
| Number of Data Items Transferred (1) | Number of Database Nodes (1) | Number of Early Field Trial Testing Non-security Faults (1) |
| Number of Executable Lines of Code (1) | Number of External Function Callers (1) | Number of External Functions Called (1) |
| Number of External User Nodes (1) | Number of Failures (1) | Number of Features (1) |
| Number of File Nodes (1) | Number of Forks (1) | Number of Function Parameters (1) |
| Number of Function Return Points (1) | Number of Function Testing Non-security Faults (1) | Number of Functions Added (1) |
| Number of Functions Deleted (1) | Number of Functions Modified (1) | Number of Future Changes (1) |
| Number of Future Different Authors (1) | Number of Global Variables (1) | Number of HTML Sinks (1) |
| Number of Hunks (1) | Number of Indirect Incoming Connections (1) | Number of Indirect Outgoing Connections (1) |
| Number of Instances of External Function Call (1) | Number of Internal Functions Called (1) | Number of Internal Use Non-security Faults (1) |
| Number of Invocations (1) | Number of Member Nodes (1) | Number of Non-security Faults (1) |
| Number of Numerical Data Conversion Nodes (1) | Number of Past Changes (1) | Number of Past Different Authors (1) |
| Number of Performance Testing Non-security Faults (1) | Number of Persistent Data Nodes (1) | Number of Predefined Return Nodes (1) |
| Number of Regression Bugs (1) | Number of Regular-expression-based Substring Replacement Nodes (1) | Number of Runtime Function Callers (1) |
| Number of Security Bugs (1) | Number of SQL Sinks (1) | Number of SQLi-sanitizing Nodes (1) |
| Number of Stability Bugs (1) | Number of Stars (1) | Number of Static Analysis Warnings (1) |
| Number of Stress Testing Non-security Faults (1) | Number of String-based Substring Replacement Nodes (1) | Number of System Testing Non-security Faults (1) |
| Number of Taintedness Propagation Nodes (1) | Number of Test Fail Bugs (1) | Number of Unclassified Nodes (1) |

Continued on next page

Table F.1. Vulnerability discovery metrics identified in the systematic literature review (Continued)

| Metric (# Primary Studies) | | |
|---|---|---|
| Number of Uninitialized Nodes (1) | Number of Unique Ex-contributors (1) | Number of Unique Incoming Connections (1) |
| Number of Unique Outgoing Connections (1) | Number of User-defined Functions (1) | Number of Vulnerabilities (1) |
| Number of XSS-sanitizing Nodes (1) | Organization Intersection Factor (1) | Overall Organization Ownership (1) |
| Package Dependency (1) | Percentage Commits (1) | Percentage of Contributors at Organization Level (1) |
| Percentage of Interactive Line Churn (1) | Positivity (1) | Programming Language (1) |
| Proposition Density (1) | Proximity to Dangerous Point (1) | Proximity to Entry Point (1) |
| Proximity to Exit Point (1) | Ratio of External to Internal Data Flow (1) | Ratio of Outgoing to Incoming Data Flow (1) |
| Release Time Lag (1) | Repeat Frequency (1) | Review Message TF-IDF (1) |
| Risky Walk (1) | Security Experience (1) | Shared Code (1) |
| Source Lines of Non-declarative Code (1) | Source Lines of Non-HTML Code (1) | Stability Experience (1) |
| Test Fail Experience (1) | Text-based Data Access (1) | Thirty Day Number of Affected Authors (1) |
| Thirty Day Percentage of Interactive Line Churn (1) | Tokens in Commit Message (1) | Tokens in Commit Patch (1) |
| | Yngve Score (1) | |

# Appendix G

# Institutional Review Board

Shown in Figure G.1 is Form C (IRB Decision Form) from the Institutional Review Board at Rochester Institute of Technology approving our proposed protocol to provide Chromium developers metrics-derived feedback on security.

R·I·T

**Rochester Institute of Technology**

RIT Institutional Review Board for the
Protection of Human Subjects in Research
141 Lomb Memorial Drive
Rochester, New York 14623-5604
Phone: 585-475-7673
Fax: 585-475-7990
Email: hmfsrs@rit.edu

**Form C**
**IRB Decision Form**
**FWA# 00000731**

**TO:** Nuthan Munaiah

**FROM:** RIT Institutional Review Board

**DATE:** **January 26, 2018**

**RE:** Decision of the RIT Institutional Review Board

Project Title – Assisted Discovery of Software Vulnerabilities

The Institutional Review Board (IRB) has taken the following action on your project named above.

☒ Exempt 46.101 (b) (2)

Now that your project is approved, you may proceed as you described in the Form A.

You are required to submit to the IRB any:

- **Proposed** modifications and wait for approval before implementing them,
- Unanticipated risks, and
- Actual injury to human subjects.

[Signature Redacted]

Heather Foti, MPH
Associate Director
Office of Human Subjects Research

Revised 08.17.2017

Figure G.1: Form C (IRB Decision Form) from the Institutional Review Board at Rochester Institute of Technology

# Appendix H

# Security Feedback Assessment Survey

Shown in Table H.1 it the survey questionnaire we used to assess Chromium developers' perception of the security feedback we provided. The survey questionnaire and the protocol associated with its administration, and the subsequent analyses of the responses, was reviewed and approved by the Institutional Review Board at Rochester Institute of Technology.

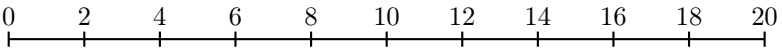
Table H.1: Survey questionnaire used to assess the Chromium developers' perception of the security feedback we provided

**Q1 of 3. Please indicate your level of agreement on the following statements about the feedback we provided.**

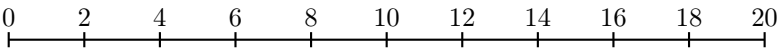
| | **Strongly Agree** | **Somewhat Agree** | **Neither Agree nor Disagree** | **Somewhat Disagree** | **Strongly Disagree** |
|---|---|---|---|---|---|
| The feedback disrupted my workflow. | ❍ | ❍ | ❍ | ❍ | ❍ |
| I read the feedback carefully. | ❍ | ❍ | ❍ | ❍ | ❍ |
| I understood the feedback that was provided. | ❍ | ❍ | ❍ | ❍ | ❍ |
| I discussed the feedback with my colleagues. | ❍ | ❍ | ❍ | ❍ | ❍ |
| The feedback was surprising to me. | ❍ | ❍ | ❍ | ❍ | ❍ |
| The feedback caused me to think about the security implications of this change. | ❍ | ❍ | ❍ | ❍ | ❍ |
| I found the feedback to be useful. | ❍ | ❍ | ❍ | ❍ | ❍ |
| I would prefer the feedback provided directly to me not in a group setting. | ❍ | ❍ | ❍ | ❍ | ❍ |

**Q2 of 3. How many years have you ...**

... been working on the Chromium project?




... been working in software development?




**Q3 of 3. Is there anything you would like to tell us?**

# Appendix I

# Sample Security Feedback

Shown in Figure I.1 is a sample security feedback that we provided to Chromium developers in a code review identified by `I0f3ee423792469a735148d9008e98341074c7d5b`.

Hello,

I'm a PhD Student at the Rochester Institute of Technology. We are working on a machine learning technique to discover vulnerabilities with metrics.

We believe 1 of the 5 source code files modified in I0f3ee423792469a735148d9008e98341074c7d5b is at a higher risk of having undiscovered vulnerabilities. The suspect file, along with evidence to support the assessment, is enumerated below.

− chrome/browser/chrome_browser_main.cc − The file has been changed a lot (churn at the 94th percentile) by many developers who also changed many other files (contribution centrality at the 100th percentile). The file is also hard to test exhaustively (nesting at the 96th percentile).

Any questions? I'm open for discussion. For more information on the metrics used in the assessment, please visit https://samaritan.github.io/metrics

We would really appreciate if you could spend 5 minutes of your time to fill out an anonymous survey at https://rit.az1.qualtrics.com/jfe/form/SV_ebrp7G3vpPZ01H7?Source=I0f3ee423792469a735148d9008e98341074c7d5b to help us improve the feedback.

Thank you,
Nuthan Munaiah

Figure I.1: Sample security feedback provided to Chromium developers in the code review identified by `I0f...d5b`

# Appendix J

# Initiation of Dialogue with Chromium Developers

Shown in Figure J.1 is the message posted to both `chromium-dev` and `security-dev` Google Groups to elicit Chromium developers' expectations from vulnerability discovery metrics.

Hello,

My name is Nuthan Munaiah and I am a PhD Student at Rochester Institute of Technology in Rochester, NY, USA. I am working on assessing the utility of vulnerability discovery metrics in assisting developers engineer secure software (See Munaiah 2018 for an overview of my research project).

Over the last few weeks, I have been using nine metrics collected from the Chromium project to derive security feedback. I started providing the security feedback as comments on code reviews but, after providing feedback on a few changes, my access to the code review system was revoked, citing spamming as the reason. One of the developers suggested that I try security−dev and/or chromium−dev Google Groups as alternative places to provide the security feedback. However, after posting two security feedbacks on the security−dev Google Group, I was accused (in the anonymous survey) of posting spam again. With the exception of the issue tracker, I seem to have exhausted potential places I could provide security feedback on.

When designing the research study, I had anticipated non−response to survey as a potential challenge. However, the overwhelming opinion that security feedback is spam has been surprising. There is a large community of academic researchers working to assist software engineers discover and resolve bugs (especially vulnerabilities) using advances in data science and software repository mining. See Morrison et al. 2018 for a survey of research in the realm of security metrics. I am reaching out the chromium−dev Google Group to take a step back and address certain overarching questions to better understand and be respectful of the Chromium development processes.

* What are your thoughts on the security metrics research community?
* Are we producing work that could be valuable to you?Do you think there is a place for security metrics in the Chromium development life cycle?
* When, in the Chromium development life cycle, do you think is an appropriate opportunity for providing metrics−derived security feedback?

Thank you,
Nuthan Munaiah
https://nuthanmunaiah.github.io/
https://samaritan.github.io/about/

References

Munaiah, Nuthan "Assisted Discovery of Software Vulnerabilities" In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pp. 464−467. ACM, 2018.
Morrison, Patrick, David Moye, Rahul Pandita, and Laurie Williams "Mapping the field of software life cycle security metrics" Information and Software Technology 102 (2018): 146−159.

Figure J.1: Message posted to both `chromium-dev` and `security-dev` Google Groups to elicit Chromium developers' expectations from vulnerability discovery metrics